HARDWARE ENTANGLED SECURITY PRIMITIVES: ATTACKS AND DEFENSES

Vom Fachbereich Informatik (FB 20)
an der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines Doktor-Ingenieurs
genehmigte Dissertation von:

MSc. Shaza Zeitouni

Referenten:
Prof. Dr.-Ing. Ahmad-Reza Sadeghi (Erstreferent)
Prof. Dr.-Ing. Nele Mentens (Zweitreferent)

Tag der Einreichung: 30. Juli 2021
Tag der Disputation: 20. September 2021

TECHNISCHE
UNIVERSITÄT
DARMSTADT

System Security Lab
Fachbereich für Informatik
Technische Universität Darmstadt

Hochschulkennziffer: D17

Shaza Zeitouni:
Hardware Entangled Security Primitives: Attacks and Defenses
Darmstadt, Technische Universität Darmstadt
Jahr der Veröffentlichung der Dissertation auf TUprints: 2022
URN: urn:nbn:de:tuda-tuprints-215527
Tag der mündlichen Prüfung: 20.09.2021

ABSTRACT

Hardware-assisted security aims at protecting computing systems against software-based attacks that can affect the different software layers. This is attained by leveraging hardware components or modules to enforce strict security measures and thus providing stronger security guarantees compared to software-only solutions. The *trusted* hardware components form together the so-called *trust anchor*, which comprises various primitives to support different security protocols and services such as authentication, platform integrity, runtime protection, trusted execution and trusted configuration, to name some.

This thesis consists of two parts: i) an offensive part, where we present our findings based on attacks we conducted on hardware-based security primitives that can be deployed in trust anchors for platform authentication and cryptographic key generation, and ii) a defensive part, where we present our novel hardware-assisted defenses/architectures for platform integrity at runtime and trusted configuration that are based on trust anchors of our design. The contributions are organized in three pivots based on the security service provided by the trust anchor.

Platform Authentication. Physically Unclonable Functions (PUFs) are hardware security primitives that leverage the innate characteristics of hardware due to its manufacturing process for the generation of *device-specific* identifiers or cryptographic keys. Therefore, PUFs have been considered as a promising cost-effective primitive/component in trust anchors for constrained embedded devices. In this part of the thesis we evaluate the security of several PUF primitives. We demonstrate a noninvasive fault injection attack on SRAM PUFs that is conducted by controlling the voltage supply to the PUF under attack for the recovery of the secret PUF response [1]. Then, we present remote software-based fault injection attack on Rowhammer PUFs and modeling attacks on Rowhammer PUFs and memristor-based PUFs that require no physical access to the PUF under attack [2, 3]. This pivot is based on the following publications:

[1] Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, Ahmad-Reza Sadeghi. "Remanence Decay Side-Channel: The PUF Case". In IEEE Transactions on Information Forensics and Security (TIFS), Vol. 11, 2015.

[2] Shaza Zeitouni, David Gens, Ahmad-Reza Sadeghi. "It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs". In Proceedings of the 55th ACM/IEEE Design Automation Conference (DAC'18), 2018.

[3] Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, Ahmad-Reza Sadeghi. "On the Security of Strong Memristor-based Physically Unclonable Functions". In Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20), 2020.

Runtime Protection. Memory corruption attacks aim at diverting the execution of software at runtime without violating its integrity at rest. While *static* attestation is a

well established approach to verify the trustworthiness/integrity of software components and detect malware attacks, it cannot detect runtime attacks. In this part of the thesis, we present our runtime defenses for embedded systems under different deployment and adversary models and their underlying hardware-based trust anchors that we design and implement. We present i) LO-FAT, the first hardware-based control-flow attestation scheme to mitigate runtime control-flow attacks [4], ii) ATRIUM, the first runtime attestation scheme to capture *executed* instructions/binaries and control-flow behavior simultaneously to mitigate runtime control-flow as well as Time of Check Time of Use attacks [5], iii) CHASE, a flexible runtime attestation scheme suitable for real-time constrained devices [6] and iv) HardScope, a runtime context-specific memory isolation scheme to efficiently mitigate currently-known runtime data-oriented attacks [7]. This pivot is based on the following publications:

[4] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, Ahmad-Reza Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In Proceedings of the 54th ACM/IEEE Design Automation Conference (DAC'17), 2017.

[5] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, Ahmad-Reza Sadeghi. "ATRIUM: Runtime Attestation Resilient Under Memory Attacks". In Proceedings of the 36th ACM/IEEE International Conference on Computer Aided Design (ICCAD'17), 2017.

[6] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, Ahmad-Reza Sadeghi. "CHASE: Configurable Hardware-Assisted Security Extension for Real-Time Systems". In Proceedings of the 38th ACM/IEEE International Conference on Computer Aided Design (ICCAD'19), 2019.

[7] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, Ahmad-Reza Sadeghi. "HardScope: Hardening Embedded Systems Against Data-Oriented Attacks". In Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19), 2019.

Trusted Configuration. Due to their flexibility and high performance-to-power ratio, Field Programmable Gate Arrays (FPGAs) have found their way into data centers. Major Cloud Service Providers (CSPs) offer their clients FPGA-accelerated compute instances and allow them to *freely configure* the FPGAs. However, this deployment model engenders a new type of physical attacks that can be launched *remotely* by clients using only *malicious* FPGA configurations. In this part of the thesis, we systematize the research work on cloud FPGAs and spot the light on fundamental security concerns and challenges [8]. Among them, the *mutual* trust problem of FPGA configuration: clients aim to protect their proprietary designs by encrypting FPGA configurations, while CSPs do not support the use of encrypted configurations and require access to FPGA configurations to inspect for malicious primitives, e.g. voltage sensors. To tackle this open challenge, we present a security protocol between the involved parties and its underlying hardware-based trust anchor that we design and implement for trusted configuration on cloud FPGAs [9]. This pivot is based on the following publications:

[8] Ghada Dessouky, Ahmad-Reza Sadeghi, Shaza Zeitouni. "SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities". In Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P'21), 2021.

[9] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, Nele Mentens. "Trusted Configuration in Cloud FPGAs". In Proceedings of the 29th IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM'21), 2021.

ZUSAMMENFASSUNG

Hardwareunterstützte Sicherheit zielt darauf ab, IT Systeme vor softwarebasierten Angriffen zu schützen, die die verschiedenen Softwareschichten betreffen können. Dies wird erreicht, indem Hardwarekomponenten oder -module genutzt werden, um strenge Sicherheitsmaßnahmen durchzusetzen und somit stärkere Sicherheitsgarantien im Vergleich zu reinen Softwarelösungen zu bieten. Die *vertrauenswürdigen* Hardwarekomponenten bilden zusammen den sogenannten *Vertrauensanker*, der verschiedene Primitive umfasst, um verschiedene Sicherheitsprotokolle und -dienste wie Authentifizierung, Plattformintegrität, Laufzeitschutz, vertrauenswürdige Ausführung und vertrauenswürdige Konfiguration zu unterstützen, um einige Aufgaben zu nennen.

Diese Dissertation besteht aus zwei Teilen: i) einem offensiven Teil, in dem wir unsere Ergebnisse basierend auf Angriffen auf Hardware-Sicherheitsprimitive präsentieren, die in Vertrauensankern für die Plattformauthentifizierung und die Generierung kryptographischer Schlüssel eingesetzt werden können, und ii) einen defensiven Teil, in dem wir unsere neuartigen hardwaregestützten Verteidigungen/Architekturen für Plattformintegrität zur Laufzeit und vertrauenswürdige Konfiguration präsentieren, die auf Vertrauensankern unseres Designs basieren. Die Beiträge sind in drei Gruppen geteilt, basierend auf der Sicherheitsdienstleistung des Vertrauensankers.

Plattformauthentifizierung. Physically Unclonable Functions (PUFs) sind Hardware-Sicherheitsprimitive, die die intrinsischen/angeborenen Eigenschaften von Hardware aufgrund ihres Herstellungsprozesses für die Generierung von gerätespezifischen Identifikatoren oder kryptographische Schlüsseln nutzen. Daher wurden PUFs als vielversprechende kostengünstige Grundelemente/Komponenten in Vertrauensankern für eingeschränkte eingebettete Geräte angesehen. In dieser Dissertation evaluieren Wir die Sicherheit mehrerer PUF-Primitiven. Wir demonstrieren einen nichtinvasiven Fehlerinjektionsangriff auf SRAM-PUFs, der durch Steuern der Spannungsversorgung der angegriffenen PUF zur Wiederherstellung der geheimen PUF-Antwort durchgeführt wird [1]. Dann präsentieren wir Software-basierte Remote-Angriffe auf die Rowhammer PUFs und Memristor-basierte PUFs, die keinen physischen Zugriff auf die angegriffene PUF erfordern [2, 3]. Diese Gruppe basiert auf den folgenden Publikationen:

[1] Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, Ahmad-Reza Sadeghi. "Remanence Decay Side-Channel: The PUF Case". In IEEE Transactions on Information Forensics and Security (TIFS), Vol. 11, 2015.

[2] Shaza Zeitouni, David Gens, Ahmad-Reza Sadeghi. "It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs". In Proceedings of the 55th ACM/IEEE Design Automation Conference (DAC'18), 2018.

[3] Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, Ahmad-Reza Sadeghi. "On the Security of Strong Memristor-based Physically Unclonable Functions". In Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20), 2020.

Plattformintegrität zur Laufzeit. Laufzeitangriffe zielen darauf ab, die Ausführung von Software zur Laufzeit umzuleiten, ohne ihre Integrität im Ruhezustand zu verletzen. Die Attestierung ist ein etablierter Ansatz, um die Vertrauenswürdigkeit/Integrität von Softwarekomponenten zu überprüfen und Malware-Angriffe zu erkennen, kann jedoch in seiner statischen Grundform Laufzeitangriffe nicht erkennen. Wir präsentieren Laufzeitverteidigungen für eingebettete Systeme unter verschiedenen Bereitstellungs- und Gegnermodellen und ihren zugrunde liegenden hardwarebasierten Vertrauensankern, die wir entwerfen und implementieren. Wir präsentieren i) LO-FAT, das erste hardwarebasierte Kontrollfluss-Attestierung zur Abschwächung von Laufzeit-Kontrollfluss-Angriffen [4], ii) ATRIUM, das erste Laufzeit-Attestierung, das sowohl ausgeführte Befehle als auch Kontrollflussverhalten meldet, um sowohl Kontrollfluss- als auch Time-of-Check-Time-of-Use-Angriffe zu mindern [5], iii) CHASE vor, ein Laufzeit-Attestierung, das für echtzeitbeschränkte Geräte geeignet ist [6] und iv) HardScope, ein laufzeitkontextspezifisches Speicherisolationsschema, um derzeit bekannte laufzeitdatenorientierte Angriffe effizient abzuwehren [7]. Diese Gruppe basiert auf den folgenden Publikationen:

[4] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, Ahmad-Reza Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In Proceedings of the 54th ACM/IEEE Design Automation Conference (DAC'17), 2017.

[5] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, Ahmad-Reza Sadeghi. "ATRIUM: Runtime Attestation Resilient Under Memory Attacks". In Proceedings of the 36th ACM/IEEE International Conference on Computer Aided Design (ICCAD'17), 2017.

[6] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, Ahmad-Reza Sadeghi. "CHASE: Configurable Hardware-Assisted Security Extension for Real-Time Systems". In Proceedings of the 38th ACM/IEEE International Conference on Computer Aided Design (ICCAD'19), 2019.

[7] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, Ahmad-Reza Sadeghi. "HardScope: Hardening Embedded Systems Against Data-Oriented Attacks". In Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19), 2019.

Vertrauenswürdige FPGA Konfiguration. Aufgrund ihrer Flexibilität und ihres Leistungsverhältnis haben Field Programmable Gate Arrays (FPGAs), die *rekonfigurierbare* Geräte sind, ihren Weg in Rechenzentren gefunden. Cloud-Dienstanbieter bieten ihren Kunden FPGA-beschleunigte Compute-Instanzen und erlauben ihnen, die FPGAs frei zu konfigurieren. Dieses Bereitstellungsmodell erzeugt jedoch eine neue Art physischer Angriffe, die von Kunden *aus der Ferne* gestartet werden können, indem sie nur böswillige FPGA-Konfigurationen verwenden. In dieser Dissertation systematisieren wir die Forschungsarbeiten zu Cloud-FPGAs und beleuchten grundlegende Sicherheitsbedenken und -herausforderungen [8]. Darunter das Problem des gegenseitigen Vertrauens bei der FPGA-Konfiguration: Kunden zielen darauf ab, ihre proprietären Designs durch Verschlüsselung von FPGA-Konfigurationen zu schützen, während Cloud-Dienstanbieter die Verwendung verschlüsselter Konfigurationen nicht unterstützen und Zugriff auf FPGA-Konfigurationen benötigen, um sie auf böswillige Primitiven zu untersuchen, z.B. Spannungssensoren. Um diese offene Herausforderung anzugehen, präsentieren wir ein Sicherheitsprotokoll zwischen den beteiligten Parteien und dem zugrunde liegenden hardwarebasierten Vertrauensanker, den wir für eine vertrauenswürdige Konfiguration auf Cloud FPGAs entwerfen und implementieren [9]. Diese Gruppe basiert auf den folgenden Publikationen:

[8] Ghada Dessouky, Ahmad-Reza Sadeghi, Shaza Zeitouni. "SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities". In Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P'21), 2021.

[9] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, Nele Mentens. "Trusted Configuration in Cloud FPGAs". In Proceedings of the 29th IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM'21), 2021.

ACKNOWLEDGMENTS

Foremost, I would like to thank my supervisor Prof. Ahmad-Reza Sadeghi for the opportunity to work at the System Security Lab in TU Darmstadt and for his guidance during my journey to pursue a PhD. This thesis is the result of collaborations with several co-authors from different universities.

Throughout the past years I had the pleasure to collaborate with distinguished professors and researchers. In particular, I would like to thank my external supervisor Prof. Nele Mentens, Prof. Thomas Schneider, Prof. Sebastian Faust, Prof. Iryna Gurevych, Prof. Dirk Koch, Prof. N. Asokan, Prof. Farinaz Koushanfar, Prof. Lucas Davi, Prof. Yier Jin, Patrick Koeberl, Christian Wachsmann, Andrew Paverd, Thomas Nyman, Jo Vliegen, Ágnes Kiss, Giulia Traverso, Daniel Demmler, Michael Zohner and many others.

I would like to thank all my friends and colleagues at the System Security Lab with whom I had joyful lunch breaks and meaningful discussions. A very special thanks goes to Ghada Dessouky and Ahmad Ibrahim for the productive discussions and collaborations. I was also very lucky for having the opportunity to make new friends in Darmstadt. My deep gratitude to Ann-Kathrin Braun for the many cheerful moments and her continuous support since the first day I arrived in Darmstadt, to my office-mate Ghada Dessouky for the great times we had in and out the office and to Ágnes Kiss and Giulia Traverso with whom we had so much fun and fruitful discussions during CROSSING retreats.

As a member of the collaborative research center CROSSING at TU Darmstadt, I would like to thank Prof. Johannes Buchmann, Prof. Marc Fischlin, Stefanie Kettler, Johannes Braun and Jacqueline Brendel for their support and guidance.

Last but not least, I am very grateful to my extended family for their love and encouragement during my life. My uttermost gratitude goes to my mother, my late father, my husband, daughters and siblings for their unlimited support and faith in me, I could not have made it without them.

CONTENTS

LIST OF FIGURES

ACRONYMS

IoT	Internet of Things
ISA	Instruction Set Architecture
RISC-V	Reduced Instruction Set Computer V
MIPS	Microprocessor without Interlocked Pipelined Stages
TPM	Trusted Platform Module
TEE	Trusted Execution Environment
SGX	Intel Software Guard Extension
CET	Intel Control-Flow Enforcement Technology
PA	ARM Pointer Authentication
CFG	Microsoft Control Flow Guard
PUF	Physically Unclonable Function
RO PUF	Ring-Oscillator PUF
APUF	Arbiter PUF
CRP	Challenge-Response Pair
SoC	System-on-Chip
CMOS	Complementary Metal Oxide Semiconductor
IC	Integrated Circuit
ASIC	Application-Specific Integrated Circuit
FPGA	Field Programmable Gate Array
CPU	Central Processing Unit
GPU	Graphic Processing Unit
SRAM	Static Random Access Memory
DRAM	Dynamic Random Access Memory
RRAM	Resistive Random Access Memory
ROP	Return-Oriented Programming
JOP	Jump-Oriented Programming
DOP	Data-Oriented Programming
DEP	Date Execution Prevention
ASLR	Address Space Layout Randomization
CFI	Control-Flow Integrity

DFI	Data-Flow Integrity
CFA	Control-Flow Attestation
DFA	Data-Flow Attestation
CFG	Control-Flow Graph
DFG	Data-Flow Graph
TOCTOU	Time of Check Time of Use
DoS	Denial of Service
CSP	Cloud Service Provider
IP	Intellectual Property
ML	Machine Learning
FaaS	FPGA-as-a-Service
AaaS	Acceleration-as-a-Service
SMPC	Secure Multi-Party Computation
GC	Garbled Circuits
GMW	Goldreich-Micali-Wigderson

1 INTRODUCTION

In a world of ubiquitous and inter-connected computing systems, establishing and retaining trust are challenging tasks, particularly in the light of the increasing attacks such as malware attacks [10, 11, 12, 13], runtime attacks [14, 15, 16, 17, 18] and software-based microarchitectural attacks [19, 20, 21]. Existing software-only defenses have been proven insufficient, since they can be bypassed by more sophisticated attacks, thereby giving rise to hardware-assisted security solutions. Consequently, there has been a paradigm shift towards hardware-assisted defenses and security architectures propelled by the necessity of stronger security guarantees and less performance overhead. During the last two decades, we have witnessed great advances in this direction with the advent and the deployment of various hardware-based security solutions into real-world products, such as Trusted Platform Module (TPM) [22], Physically Unclonable Function (PUF) [23], ARM TrustZone [24], Intel Software Guard Extension (SGX) [25] and Intel Control-Flow Enforcement Technology (CET) [26].

In general, hardware-assisted security solutions imply the existence of an immutable root of trust, or simply a *trust anchor*, in the hardware of a computing system in order to support cryptographic protocols, e.g. authentication, and other security services, e.g. platform integrity, runtime protection [27, 28, 29, 30, 31], trusted or isolated execution [25, 24, 32, 33, 34, 35, 36], trusted configuration [37, 38] and secure storage.

In the following we discuss trust anchor's components in Section 1.1 and highlight the security services that are addressed in this thesis in Section 1.2. Then, we present our main contributions in Section 1.3 and further contributions that are not included in this thesis in Section 1.4. Finally, we outline the upcoming chapters in Section 1.5.

1.1 TRUST ANCHOR DESIGN SPACE

Trust Anchor Components. A trust anchor may comprise a variety of hardware primitives and building blocks, e.g. a cryptographic co-processor, cryptographic engines/accelerators, a random number generator, a key generation scheme, secure persistent storage and/or an access control enforcement logic, depending on the supported security services. Note that for cryptographic key storage in a trust anchor, two approaches are feasible: i) a secure non-volatile memory[1] that stores the cryptographic key permanently or ii) a PUF-based key generation scheme that (re)generates the cryptographic key on-demand.

PUFs leverage unique hardware characteristics that ensue from the uncontrollable variation during Integrated Circuits (ICs) manufacturing process to generate *device-specific* identifiers or cryptographic keys. They can be built with dedicated hardware components, e.g. the Arbiter PUFs [39, 40, 41], or by leveraging existing hardware

1 Non-volatile memory does not lose its content when powered off.

components found in any computing system, such as the Static Random Access Memory (SRAM) and the Dynamic Random Access Memory (DRAM), e.g. SRAM PUFs [42, 43] and DRAM PUFs [44, 45, 46]. Owning to their envisioned properties, i.e. uniqueness, reliability, unpredictability, and tamper-evidence, PUFs have been perceived as a promising cost-effective replacement of secure non-volatile memory and further deployed in various security protocols such as authentication. We present an overview on the state-of-the-art on PUFs in Chapter 2.

In this thesis, we evaluate the security of several PUFs designs, as security primitives deployed in trust anchors, for cryptographic key generation and authentication.

Trust Anchors in Computing Systems. Figure 1 shows the typical components that can be found in embedded, Internet of Things (IoT) or edge, devices or in cloud machines. Note that some software stack components can be omitted/added based on the deployment model and requirements. In such environments an adversary that can compromise the entire software stack,[2] i.e. firmware, hypervisor, operating system and applications, is a reasonable assumption [24, 25]. Therefore, to protect computing systems from software-based attacks, a trust anchor is a prerequisite to achieve certain security guarantees in the system.

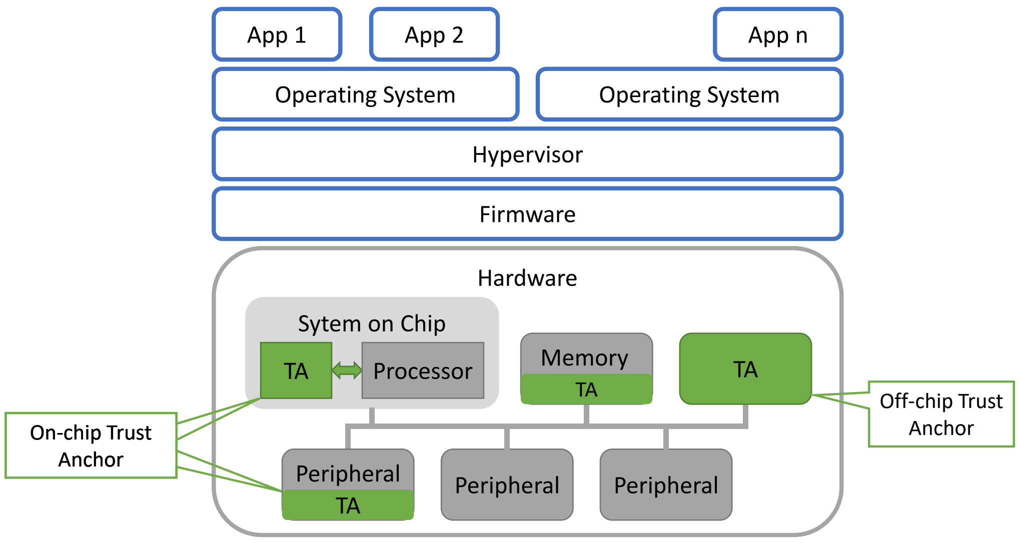


Figure 1: Trust Anchors (TAs) in a Computing System.

In Figure 1 we show the different possible positions of trust anchors in the platform. A trust anchor can be i) a stand-alone chip, e.g. a discrete TPM [22], ii) integrated with the processor in a System-on-Chip (SoC) or iii) integrated with the memory or peripheral devices, which vary from simple input/output, storage and communication devices to specialized compute units, e.g. a Graphic Processing Unit (GPU) or a Field Programmable Gate Array (FPGA). Thus, the design of the trust anchor would be dependent on the

2 Except for a small piece of software that is assumed in some security architectures to be trusted and protected by the trust anchor.

security requirements. For example, a trust anchor can be needed to simply provide secure storage or secure communication. In more complex peripherals such as the FPGA, which is a software-configurable hardware device, an on-chip trust anchor would be needed to ensure benign FPGA configuration. Next, we briefly describe some of the security services that can be supported by a trust anchor.

1.2 SECURITY SERVICES

In this section we highlight the security services that we focus on in this thesis and show them in Figure 2, where security services supported by our novel trust anchors are shown in gray.

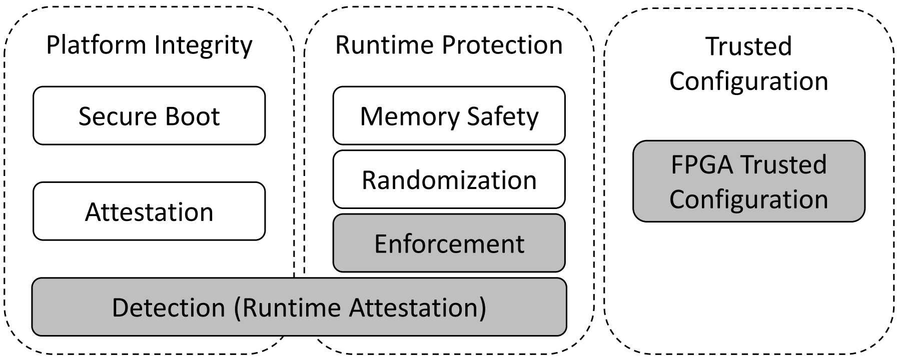


Figure 2: Security Services supported in this Dissertation.

Platform Integrity. Secure boot and attestation are two prominent and complementary approaches that are widely adopted to protect the integrity of a platform and mitigate malware attacks. Secure boot ensures that only trustworthy software components are loaded and executed when the system is booted. This is achieved gradually by verifying the authenticity/integrity of each component in the boot sequence before its execution. Thus, secure boot aims to prevent malware from being loaded and executed and provides assurance about the initial state of a platform after power up. On the other hand, attestation allows a trusted entity, the verifier, to examine the status of a platform by verifying the authenticity and integrity of its memory content. Attestation requests can be sent at any point in time defined by the verifier. Thus, attestation enables the detection of malware presence on the platform. Both, secure boot and attestation, require the presence of a trust anchor to verify or compute the proof of authenticity/integrity [22, 47]. While both methods contribute to the protection of software integrity, they give no guarantees on software execution integrity, i.e. no guarantees that software is executed correctly at runtime.

Runtime Protection. Runtime attacks leverage vulnerabilities in a software code, e.g. buffer overflows, during its execution. An attacker exploits a bug in the victim code in order to gain control over its execution and trigger malicious actions that are not intended by the software developer. These stealthy attacks aim at diverting the execution of a software code at runtime only without violating its integrity at rest. Runtime attacks can be roughly categorized into control-flow attacks [14, 15, 16] and data-oriented attacks [17, 18]. To mitigate runtime attacks, different concepts have been thoroughly investigated at different stages of a runtime exploit. Memory safety solutions for type-unsafe programming languages aim at preventing memory corruption and thus thwarting the first stage of a runtime exploit. Complementary defenses, whether randomization-based [27, 48] or enforcement-based [49, 50, 30, 51], hamper the second stage of a runtime exploit by impeding the execution of malicious actions, e.g. control-flow hijacking. Nevertheless, the persistent nature of runtime attacks has led to a continuous arms race between defenses and more sophisticated attacks. We present a brief overview on the current state of runtime attacks and defenses in Chapter 3.

In this thesis, we further follow along a recent line of research that rather aims at the detection of runtime attacks [52] such that execution details are reported to a trusted entity through remote attestation to verify the execution integrity.

Trusted Configuration. Reconfigurable computing devices such as FPGAs are ICs that can be electrically programmed by end users using binary files. The flexible nature of FPGAs and their toolchains allow users to implement various digital circuits on the FPGA fabric including tiny voltage and temperature sensors [53, 54]. FPGAs can operate as stand-alone computing systems or be integrated as hardware accelerators in more complex systems. When deployed in untrusted environments, the confidentiality and integrity of FPGA configuration can be protected with the help of a trust anchor [37, 38] provided by FPGA vendors and hard-coded on the FPGAs. Nevertheless, the user, i.e. the owner of the FPGA, is responsible for enabling the protection of designs on the FPGA. That is, the user must provision the keys on the FPGA's trust anchor securely before it is deployed in-field. Recently, commodity FPGAs have been deployed in datacenters in a temporal multi-tenant sharing model [55, 56], where clients can *rent* FPGA-accelerated compute instances for *some time* and *freely configure* the FPGA with own (malicious) functionalities, which can have serious impact on the cloud infrastructure [57, 58]. On the other hand, design protection on cloud FPGAs is not supported and clients are forced to divulge their proprietary designs to the Cloud Service Provider (CSP), which may violate Intellectual Property (IP) policies for companies. We present a short summary on the recent trend of FPGA-based cloud computing in Chapter 4.

In this thesis, we evaluate the security ramifications of deploying commodity FPGAs in the cloud and tackle their mutual trust challenge.

1.3 MAIN CONTRIBUTIONS

Goal and Scope of this Dissertation. The main scope of this dissertation is the design of trust anchors for hardware-assisted security architectures. First, we address one of the basic primitives/components of a trust anchor, the PUFs. In particular, we revisit the security of existing PUF designs and their pitfalls under the assumed deployment and adversary models. Then, we present novel hardware-assisted schemes for platform integrity at runtime and trusted configuration and demonstrate how the underlying trust anchors that we design and implement achieve the required security guarantees. Specifically, we advance state-of-the-art in runtime protection with novel hardware-based runtime enforcement and attestation schemes for embedded systems under different deployment and adversary models. We further explore security challenges and opportunities of the recent trend of cloud FPGAs and advance state-of-the-art with a scheme to establish trust on cloud FPGAs. The main contributions of this dissertation are presented in the upcoming Chapters 2, 3 and 4 as follow:

Security Evaluation of Physically Unclonable Functions. We focus in this part of the thesis on two essential characteristics of PUFs, reliability and unpredictability, and demonstrate attacks on different PUFs, namely, the SRAM PUF, the Rowhammer PUF and a set of hybrid memristor-based PUFs.[3] The attack on the SRAM PUF is a noninvasive fault injection attack that requires a precise control of the voltage supply to the device embedding the PUF for secret response recovery [1] (Appendix A). On the other hand, the attack on the Rowhammer PUF is a remote software-based fault injection that causes the generation of faulty PUF responses and thus faulty keys [2] (Appendix B). Finally, we present modeling attacks on the Rowhammer PUF [2] (Appendix B) and a set of memristor-based PUFs [3] (Appendix C) with the goal to build software models of the PUFs under attack that can predict the PUF behavior with high probability. Our findings are discussed in Chapter 2.

Hardware-assisted Runtime Protection. We present novel hardware-based runtime attestation and enforcement schemes that are based on trust anchors that we design and implement. Our runtime attestation schemes leverage hardware components to capture and process the runtime behavior of the attested program and are opted for different deployment scenarios and adversary models. We consider attackers with physical access to the prover device as well as remote software-only attacks. Specifically, we present i) LO-FAT, the first hardware-based runtime scheme to mitigate runtime control-flow attacks [4] (Appendix D), ii) ATRIUM, the first runtime attestation scheme to capture the executed instructions and control-flow behavior simultaneously to mitigate runtime control-flow as well as Time of Check Time of Use (TOCTOU) attacks [5] (Appendix E), iii) CHASE, a flexible runtime attestation scheme suitable for real-time constrained devices [6] (Appendix F) and iv) HardScope, a runtime enforcement scheme for memory isolation to efficiently mitigate currently-known runtime data-oriented attacks [7] (Appendix G). Our hardware-based runtime attestation and enforcement schemes and their trust anchors are presented in Chapter 3.

Hardware-assisted Trusted Configuration. We explore the security challenges and opportunities of deploying commodity FPGAs in cloud and datacenters and provide a comprehensive anatomy of the emerging threats, including state-of-the-art remote physical attacks that leverage the configurable nature of FPGAs, and potential defenses. We also discuss lessons learned from CPU-based trusted computing and draw potential analogies to FPGA-based trusted computing [8] (Appendix H). We further identify the mutual trust challenge between clients and CSPs: clients require to protect their proprietary designs,

3 A memristor is an emerging nano-technology circuit element.

while CSPs do not support the use of encrypted FPGA configurations. In fact, CSPs require access to FPGA configurations to inspect for malicious primitives prior to configuration. To tackle this problem, we present a security protocol to establish mutual trust between CSPs and clients that is based on a hardware-based trust anchor that we design and implement for cloud FPGAs [9] (Appendix I). Thus, paving the way to isolated/trusted execution on cloud FPGAs. Our work is presented in Chapter 4.

1.4 FURTHER CONTRIBUTIONS

Other contributions that are not included in this dissertation:
Secure Multi-Party Computation (SMPC). SMPC allows multiple parties to evaluate a function on private inputs revealing only the result of the computation. The two prominent protocols for SMPC, Yao's Garbled Circuits (GC) and the protocol of Goldreich-Micali-Wigderson (GMW), require the function to be evaluated in the form of a Boolean circuit. While building functionally-correct circuits for simple functions can be done manually by experts, this task is time-consuming and error-prone for larger functions. In this direction, hardware synthesis is a well-established line of research and is therefore an intuitive choice for Boolean circuits generation. Hardware synthesis tools have been first used for the purpose of generating Boolean circuits for Yao's GC in [59]. We contributed further to the research of enabling highly practical and efficient SMPC protocols in [60, 61, 62, 63]. Hardware synthesis tools primarily target hardware platforms, such as Application-Specific Integrated Circuits (ASICs) or FPGAs. Therefore, we need to customize these tools to generate Boolean circuits for specific SMPC protocols. This approach promises accelerated and scalable circuit generation, while maintaining the efficiency of hand-optimized circuits. I focused in [60, 61, 62, 63] on the design and implementation of various hardware primitives/functions including basic arithmetic operations, functions based on floating-point operations and customized MIPS-based cores. Further, I worked on the customization of hardware synthesis tools (commercial and academic) for the generation of the final Boolean circuits.

Practical Long-term Secure Distributed Storage Systems. Secret sharing-based distributed storage is one approach to provide long-term protection against quantum adversary. However, it is considered an impractical solution, since it requires establishing an information-theoretically secure channel between any two storage nodes. Not to mention the need for long-term confidential commitment schemes that are computationally impractical for large files. To mitigate the aforementioned limitations, we worked in [64] on a secret sharing-based secure distributed storage system that leverages Trusted Execution Environment (TEE) for shares generation and renewal. In this work, I contributed with the co-authors to the discussions of the core idea and the design of SAFE's protocols: Share, Renew and Reconstruct.

Non-interactive Attestation. Existing attestation schemes are vulnerable to Denial of Service (DoS) attacks through fake attestation requests sent by malicious entities. In [65], we proposed the first non-interactive attestation protocol that successfully mitigates DoS attacks. Designing such a protocol is non trivial, since it relies on an untrusted prover to initiate the attestation process. The resulting protocol is particularly suitable for low-end constrained embedded devices, since it is highly efficient in terms of power consumption and communication. I contributed in this work to the core idea of mitigating DoS attacks against remote attestation and the design of the trust anchor that triggers and computes the attestation reports.

1.5 THESIS OUTLINE

The main contributions of this dissertation are presented next. In Chapter 2 we present and discuss our results on the security evaluation of various PUF designs. Then, we present our trust anchors for runtime protection in Chapter 3 and for trusted configurable cloud computing in Chapter 4. Finally, in Chapter 5 we conclude the dissertation and refer to future research directions on PUFs and the design of trust anchors for runtime attestation and trusted configurable computing.

2 PHYSICALLY UNCLONABLE FUNCTIONS (PUFS)

My Contributions. This chapter is based on the results of three papers published in the IEEE Transactions on Information Forensics and Security (TIFS) journal and the Design Automation Conference (DAC):

[1] Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, Ahmad-Reza Sadeghi. "Remanence Decay Side-Channel: The PUF Case". In IEEE Transactions on Information Forensics and Security (TIFS), Vol. 11, 2015.

This publication is an extension of a previous work conducted by Yossef Oren and Christian Wachsmann [66]. I conducted the new experiments of the voltage drop attack, worked on the results analysis and led the submission and publication of the manuscript.

[2] Shaza Zeitouni, David Gens, Ahmad-Reza Sadeghi. "It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs". In Proceedings of the 55th ACM/IEEE Design Automation Conference (DAC'18), 2018.

I am the lead author of this work. I contributed with the co-authors to the idea of the denial of service attack on the Rowhammer PUF and co-supervised the HiWi student Mohamed Saad, who implemented part of the attacks. Further, I contributed to the modeling attack idea of the Rowhammer PUF and conducted the experiments and results analysis.

[3] Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, Ahmad-Reza Sadeghi. "On the Security of Strong Memristor-based Physically Unclonable Functions". In Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20), 2020.

I contributed with Emmanuel Stapf to the discussion of the main idea of this work. I led the work and worked on the selection of the PUF circuits evaluated in this work as well as their implementation and fine-tuning. All co-authors contributed to the selection of the machine learning algorithms used in this work. Emmanuel Stapf and Hossein Fereidooni contributed to the implementation and evaluation of the machine learning algorithms.

Chapter Outline. In the following, we introduce silicon PUFs, their properties, classes and designs in Section 2.1. Then, we provide an overview of state-of-the-art attacks including our contributions in Section 2.2. Finally, we present hybrid PUFs that leverage emerging technologies and our attacks on hybrid PUFs in Section 2.3.

2.1 PUFs DESIGN SPACE

PUFs are noisy functions that are stimulated with an input, a *challenge*, to produce an output, a *response*, that strongly depends on both the challenge and the innate physical properties of the device embedding the PUF. Silicon PUFs leverage the inevitable effect of process variation during the manufacturing of Integrated Circuits (ICs) as a source of entropy to derive a *reproducible* device-specific behavior. Given the nominal specifications for IC components, i.e. transistors and interconnect, process variation is the deviation of the resulting parameter values during IC manufacturing from the given nominal values [67]. In this section we briefly present envisioned PUFs properties, classes and deployment models.

Properties of PUFs. Researchers have investigated PUF properties that should be satisfied for the deployment in cryptographic protocols [68, 69, 70, 71]. These properties are uniqueness, reliability, unpredictability, and tamper-evidence among many others. Uniqueness implies that responses resulting from evaluating the same challenge on different PUF instances of the same family should be different, while reliability indicates the consistency of PUF responses to the same challenge that are generated under varying operating conditions, i.e. ambient temperature and supply voltage. On the other hand, unpredictability indicates that even after observing a polynomial set of Challenge-Response Pairs (CRPs) of a PUF instance, responses of new challenges remain sufficiently random. Owning to their properties and their cost-effectiveness, PUFs have been envisaged as a root of trust in ICs for the generation of fingerprints and secret keys and are further deployed in various cryptographic protocols, e.g. authentication and attestation protocols. In our attacks, we focus on two security properties of PUFs, reliability and unpredictability.

Challenge-Response Space & PUF Deployment. Depending on their input-output space, PUFs have been categorized into *strong* PUFs and *weak* PUFs [72, 73].

A **strong PUF** has an exponential number of CRPs, with respect to the number of its components, such that within a bounded amount of time, it is not feasible to measure all the CRPs. As defined by Rührmair et al. [72] a PUF is labeled ”strong“ if it also satisfies the unpredictability property. Therefore, no protection mechanisms are applied to their CRPs [72]. Strong PUFs are mainly designated for authentication protocols. The simplest, i.e. light-weight, PUF-based authentication protocol is a challenge-response protocol, where unprocessed CRPs of a PUF are sent in plain-text between the verifier and the prover. In [74, 75, 76] a survey and security analysis of PUF-based authentication protocols is presented.

A **weak PUF**, on the other hand, can only produce a limited number of responses, linear in the number of its components, and therefore are mainly deployed for secret key generation. However, since PUFs are noisy functions and their responses might not be sufficiently random, further post-processing steps are needed to generate secret keys [77, 78, 79, 80] for deployment in cryptographic protocols [81, 82]. Nevertheless, the use of weak PUFs in light-weight authentication protocols under certain constraints has been proposed [83, 84, 46].

PUF's Underlying Components. Based on the PUF' underlying components and architecture, two major groups of *silicon* PUFs can be identified: *memory-based* PUFs and *delay-based* PUFs.

Memory-based PUFs leverage the process variation in memory technologies that leads to random strength mismatch of transistors within memory cells, such as SRAM PUFs [85, 42, 86, 87] and DRAM PUFs [44, 45, 46]. In general, memory-based PUFs, can only produce a limited number of responses, linear in the number of the utilized memory cells. Therefore, they are classified as *weak* PUFs.

Delay-based PUFs [88] leverage delay differences between two identical signal propagation paths in a circuit. The delay differences are caused by wire and transistor mismatch in the two paths. The two prominent delay-based PUFs are Ring-Oscillator PUFs (RO PUFs) [89] and Arbiter PUFs (APUFs) [88].

2.2 PUFs ARMS RACE (SELECTED)

Since their introduction, PUFs have been heavily under various attacks. The attacks range from physical, which can be further categorized into (semi-) invasive and noninvasive attacks, to software-based modeling attacks where no physical access to the PUF is needed.

2.2.1 *Physical Attacks*

Although PUFs have been considered to be tamper-evident, it has been shown that PUFs are vulnerable to all kinds of physical attacks, including those that do not affect the PUF's functionality after the attack. Physical attacks target different PUF families and are highly dependent on the PUF implementation and the process technology. These attacks aim at extracting the device-specific parameter values to emulate the PUF behavior.

Noninvasive Attacks aim at extracting a PUF secret key or behavior through side-channel measurements, i.e. electromagnetic emission [90, 91] and power consumption [92], or via controlled fault injection [66, 1, 2]. Noninvasive attacks target either the post-processing circuit [90] or the PUF circuit, e.g. delay-based PUFs [91, 92] and memory-based PUFs [66, 1, 2].

In this direction we present in [1] a noninvasive fault injection attack on SRAM PUFs – SRAM PUFs leverage the start-up values of SRAM cells at power up before initializing them to defined values [85, 42]. The SRAM memory used by the PUF is assumed to be shared, after evaluating the PUF response and initializing the memory to predefined values, with other (malicious) processes on the system, which is a common assumption on constrained devices [93, 94, 95]. The attack requires a precise control of the voltage supply to the device embedding the PUF in order to induce the remanence decay effect in the SRAM cells.[1] The attack starts by writing a known pattern to SRAM memory, e.g. all-ones or all-zeros. Next, the attacker reduces the supply voltage of the device for a fixed period of time then restores the nominal operating voltage and captures the faulty PUF behavior, e.g. by using the faulty PUF response to encrypt a known message. These steps are repeated until the supply voltage is gradually reduced to zero. Thus, in each experiment more SRAM cells will decay and revert into their start-up values until all SRAM cells reach their start-up values in the final experiment. Then using differential fault analysis [96] and the resulting cipher-texts from the different experiments, the SRAM start-up value and consequently the original PUF response can be recovered. This work extends on the work by Oren et al. in [66], which leverages the remanence decay effect of SRAM memory by completely powering off the SRAM cells for increasing periods of time until they all lose their contents and reach their start-up values. The details of our attack are presented in Appendix A.

Noninvasive attacks typically require physical access or close proximity to the device under attack to measure the side-channel leakage or control the operating conditions. Nevertheless, we show that remote software-based fault injection attacks on PUFs are also feasible. In this context, we demonstrate in [2] a remote DoS attack on the Rowhammer PUF [45], which is a variant of the runtime-accessible decay-based DRAM PUFs. Rowhammer PUFs leverage the unique behavior of DRAM cells under two effects, the Rowhammer[2] and the decay effects, in DRAM memories. Given that the Rowhammer PUF is assigned a dedicated DRAM region that is only accessible by a trusted software for the purpose of querying the PUF, our attack aims at modifying the PUF responses without accessing the PUF region. This is achieved by hammering the borders of the PUF region at the time of querying the PUF response, i.e. a malicious process repeatedly accesses the rows above and below the PUF region, while the PUF response is evaluated. The attack is also applicable to decay-based DRAM PUFs [46]. Our results show the non-negligible deviation of the resulting PUF responses from the reference values. More details on the attack are available in Appendix B.

Semi-Invasive Attacks on ICs containing PUF circuits have also been demonstrated to be feasible. These attacks are conducted from the IC backside and require depackaging of the chip and thinning of the silicon substrate, then further techniques can be applied to extract PUF parameters. For example, using laser probing [97] and spatial photonic emission analysis [98] to read the start-up state of SRAM memory, laser voltage probing and imaging [99] to characterize a RO PUF, and temporal photonic emission analysis [100, 101] to measure internal paths' delays of an APUF.

2.2.2 *Software-based Modeling Attacks*

Modeling attacks aim at deriving a numerical model of the PUF under attack using a set of measured CRPs. If the derived model predicts the responses to unseen challenges with

1 SRAM cells do not lose their contents immediately when the voltage supply is off or reduced, however they decay slowly over time.

2 Excessively accessing DRAM memory cells leads to bit flips in physically adjacent memory cells.

high probability, the unpredictability of the PUF, and therefore its security, are broken. Most of existing modeling attacks leverage Machine Learning (ML) algorithms to build a PUF model [39, 102, 103, 104, 105, 106, 107, 108, 109, 110]. However, other methods that leverage the inherent noisy PUF responses by measuring each CRP several times, e.g. the differential measurements method and the least mean square method, have also been shown to be successful in [111].

During the last two decades, several improved PUF designs [39, 40, 41, 112, 113], and novel PUF designs [114, 115, 116, 117, 106] have been introduced as modeling-resilient PUFs, but were later proven to be susceptible to modeling attacks [103, 104, 106, 108, 118]. Further, several mitigation methods to modeling attacks have been introduced: i) obfuscation of PUF responses only or challenges and responses using hash functions [119] or reverse fuzzy extractors [120], ii) randomization of challenges [121], iii) concealing bits of the PUF response (Slender PUF) [122, 123] or challenge [124], iv) using finite state machine [125], v) irreversible reconfiguration of PUF behavior (Reconfigurable PUF) [126] and vi) erasing used CRPs (Erasable PUF) [127]. Nevertheless, most of these methods have been shown to be ineffective in face of modeling attacks [105, 76].

Modeling attacks require a large number of CRPs for learning and validating a PUF model. Therefore, they typically target strong PUFs with exponential CRPs. Weak PUFs, on the other hand, have been thought to be out of modeling attacks reach due to their limited CRP space. However, RO PUFs have been shown to be prone to modeling attacks [102, 103].

In this context, we also tested the applicability of modeling attacks to another weak PUF, the Rowhammer PUF. Decay-based DRAM PUFs have been proposed for the deployment in light-weight authentication protocols under certain conditions regarding the decay time. Our results indicate that the Rowhammer PUF is not secure for deployment in authentication protocols where PUF responses are exchanged in plain-text even when the decay time constraints are met. We present our modeling attack that leverage standard interpolation algorithms in Appendix B.

Hybrid Modeling Attacks leverage physical aspects of PUF implementations in combination with modeling attacks to reduce their complexity in terms of the number of required CRPs and learning time, in case ML algorithms are used. State-of-the-art hybrid attacks are either passive, i.e. use side-channel information, such as power consumption and timing information [128, 129] or active, i.e. induce faults in PUF circuits or responses through laser beam [130] or by manipulating their nominal operating conditions [129, 131, 132] to generate faulty responses.

2.3 HYBRID PUFS

This continuous arms race has driven the research community to investigate other sources of entropy for PUF designs. In recent years and due to the prominent and continuous progress in material science, novel nano-devices have been developed for beyond-silicon applications. Examples of such emerging nano technologies are carbon nanotube field-effect transistors [133], spintronic-logic devices [134], memristors [135] and many others.

A memristor is a circuit element with a dynamic resistance behavior that depends on the properties (direction and strength) of the voltage applied at ports of the memristor [136]. When no voltage is applied, the memristor maintains its most recent resistance state. Due to their compatibility with CMOS manufacturing technology, the deployment of memristive devices into different digital circuits has been thoroughly investigated [137] including memory technologies, i.e. Resistive Random Access Memory (RRAM) [137, 138], and neuromorphic computing [139]. Memristors exhibit stochastic switching behavior, i.e. a cycle-to-cycle variation when switching between low and high resistance states, that can be leveraged in addition to cell-to-cell variation, i.e. variation during the manufacturing process, as an inherent source of randomness to enable the construction of different light-weight security primitives: random number generators [140, 141] and PUFs.

Modeling Hybrid Memristor-based PUFs. Compared to existing CMOS-only PUFs, hybrid PUFs have promised higher reliability and unpredictability [142, 143, 144]. Recently, several hybrid PUF designs have been proposed, including *weak* PUFs [145, 146, 142] and *strong* PUFs [143, 147, 148, 144]. These hybrid PUF designs are mainly evaluated with circuit simulators. While many new hybrid PUF designs have been proposed, verifying their properties and security guarantees is highly challenging due to the lack of open specifications. We reproduced a set of hybrid PUFs based on memristors [149, 150, 151] to investigate their unpredictability property. In our attacks we use different ML algorithms: logistic regression, ensemble classifiers and recurrent neural networks. Our results indicate that the inspected PUFs and their XOR-based versions are still vulnerable to modeling attacks that leverage advanced ML techniques. Therefore, it is obvious that the construction of PUFs that are resilient to advanced modeling attacks is still an open challenge. More details on the modeled hybrid PUFs and our attacks are available in Appendix C.

3 HARDWARE-ASSISTED RUNTIME PROTECTION

My Contributions. This chapter is based on the results of four papers published in the Design Automation Conference (DAC) and the International Conference On Computer Aided Design (ICCAD):

[4] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, Ahmad-Reza Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In Proceedings of the 54th ACM/IEEE Design Automation Conference (DAC'17), 2017.

I contributed with Ghada Dessouky and Thomas Nyman to the design discussions and security analysis that led to this publication. I focused on the implementation of the modules that process and encode the execution metadata and the modules that control the computation of the final attestation report. Ghada Dessouky led the work and focused on the implementation modules that capture and track the execution from the processor pipeline. Andrew Paverd contributed to the discussions on the security of the scheme and Patrick Koeberl contributed to the discussions on the hardware architecture.

[5] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, Ahmad-Reza Sadeghi. "ATRIUM: Runtime Attestation Resilient Under Memory Attacks". In Proceedings of the 36th ACM/IEEE International Conference on Computer Aided Design (ICCAD'17), 2017.

I contributed with Ghada Dessouky to the design discussions and implementation that led to this publication. I led this work and focused on the implementation of the proposed scheme, while Ghada Dessouky focused on the implementation that interfaces and integrates the scheme with the processor pipeline. Orlando Arias and Dean Sullivan contributed to the attacks on state-of-the-art attestation schemes (SMART & C-FLAT). Ahmad Ibrahim contributed to the discussions on the security guarantees of the scheme.

[6] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, Ahmad-Reza Sadeghi. "CHASE: Configurable Hardware-Assisted Security Extension for Real-Time Systems". In Proceedings of the 38th ACM/IEEE International Conference on Computer Aided Design (ICCAD'19), 2019.

I co-led the work with Ghada Dessouky and contributed to the discussions on the design and implementation that resulted in this publication. I focused on enabling the attestation mechanism for securing timing-critical applications. Ghada

Dessouky focused on enabling a consolidated security extension that is configured to adapt to different security requirements and deployment settings. Ahmad Ibrahim contributed to the discussions on the security guarantees and analysis of the scheme.

[7] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, Ahmad-Reza Sadeghi. "HardScope: Hardening Embedded Systems Against Data-Oriented Attacks". In Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19), 2019.

I contributed with Ghada Dessouky, Thomas Nyman and Aaro Lehikoinen to the discussions on the design and implementation that led to this publication. Thomas Nyman led the work and conceived the idea of the Run-time Scope Enforcement (RSE) and the design of the RSE Instruction Set Extension. Aaro Lehikoinen adapted the RSE Instruction Set Extension to RISC-V and implemented the support for the new instruction in GCC. I focused on the design and implementation of the HardScope hardware extension and evaluated the overheads of the hardware extension. Ghada Dessouky focused on the design and integration of the RISC-V Instruction Set Extension and the RSE HardScope hardware extension in the processor. Kesara Gamlath and Rangana De-Silva, under Thomas's supervision, ported the implementation on an FPGA and evaluated its performance. Thomas Nyman implemented the platform software support for HardScope to the processor software stack and evaluated its security.

I focused on the design of hardware trust anchors and leveraging them to establish or enable the verification of platform runtime integrity by providing different security services under different deployment and adversarial assumptions for embedded devices. Co-author Ghada Dessouky focused on the capabilities driven from leveraging existing processor (micro-)/architectural features, extensions and trusted hardware assumptions to enable more efficient protection for software with stronger security guarantees than software-based solutions.

Chapter Outline. In the following, we introduce runtime attacks and their different classes in Section 3.1. Then, we provide an overview of state-of-the-art runtime defenses including our work in Section 3.2. Finally, we present state-of-the-art runtime attestation schemes including our hardware-based runtime attestation schemes in Section 3.3.

3.1 RUNTIME ATTACKS

Runtime attacks aim at diverting the execution of software at runtime without violating its integrity at rest. Runtime attacks, also known as memory corruption attacks, exploit vulnerabilities in a software code, e.g. buffer overflows, to gain control over its execution and perform malicious actions that are not originally intended. The different objectives of such attacks include privilege escalation, disclosure of confidential information or evasion of security countermeasures. Existing runtime attacks are mainly grouped into *control-flow* attacks and *data-oriented* attacks.

Control-Flow Attacks leverage vulnerabilities in the code of a victim program to corrupt *control data*, i.e. code pointers, such as function pointers and return addresses, that resides in the program's memory at runtime, and pervert program's execution from its intended control flow. These attacks are further classified into: i) code injection attacks [152] that require the injection of the malicious code in the data segment within the program's memory and connecting the original code to the malicious code, which is achieved by tampering with a code pointer through exploiting a program vulnerability, and ii) code-reuse attacks that utilize pre-existing executable code that is already residing in the memory rather than injecting new code. Examples of code-reuse attacks are return-into-libc attacks [153, 14], which redirect the execution into security-critical library functions, Return-Oriented Programming (ROP) [154] and Jump-Oriented Programming (JOP) [16]. ROP and JOP attacks enable the execution of arbitrary malicious code by stitching together instruction sequences that already reside in the address space of the program such that the execution is directed from one sequence to the next sequence using return and indirect branch instructions, respectively.

Data-oriented Attacks, on the other hand, are stealthier runtime attacks that divert the execution of a victim program by corrupting *non-control data*, e.g. data variables or data pointers, however, without violating the program's control flow [17]. Non-control data attacks have been proven to contrive Turing-complete Data-Oriented Programming (DOP) attacks [18]. Thus, they enable the attacker to execute arbitrary malicious operations by corrupting non-control data only to stitch together sequences of instructions while still abiding to the intended program's control flow.

3.2 ARMS RACE (SELECTED)

Several concepts have been thoroughly investigated to mitigate runtime attacks at the different phases of a runtime exploit. Memory safety solutions for type-unsafe programming languages work as a first line of defense and aim at preventing a memory corruption and thus thwarting the first phase of a runtime exploit. Such defenses include the insertion of stack or heap canaries [155, 156, 157, 158, 159] and the deployment of pointer bounds checks [160, 161, 162, 163] or pointer integrity/authentication schemes [164, 165, 166].

Complementary defenses work at preventing the second phase of a runtime exploit, i.e. mitigating the effect of exploiting a memory vulnerability, by impeding the execution of malicious actions, e.g. control-flow or data-flow hijacking. Complementary defenses can be roughly categorized into i) randomization-based solutions [167] including the widely-adopted Address Space Layout Randomization (ASLR) [27, 48], and ii) enforcement-based solutions, such as shadow stack for return addresses [28], Control-Flow Integrity (CFI) based on the program's Control-Flow Graph (CFG) [50, 29], Data-Flow Integrity (DFI) based on the program's Data-Flow Graph (DFG) [30, 31] and data-flow isolation [51, 7].

Data-Flow Isolation. In general, data-flow isolation schemes enforce memory access control policies to mitigate runtime attacks. HDFI [51] is an Instruction Set Architecture (ISA) extension that is based on memory tagging of data for access control enforcement. HDFI uses single-bit tag per memory location to distinguish between sensitive and non-sensitive data and thus supports only two protection domains/contexts simultaneously.

In this direction, we present HardScope [7] a novel ISA extension to effectively block current data-oriented attacks including DOP attacks [17, 18]. HardScope provides fine-grained context-specific memory isolation by enforcing compile-time access control policies, e.g. variable visibility rules, on every memory access, i.e. load and store, instruction at runtime. To instrument the program code, our instrumentation tool in the compiler adds HardScope instructions at specific locations in the binary depending on the required granularity and the number of execution contexts, i.e. protection domains. The inserted HardScope instructions configure at runtime our HardScope hardware components to define and terminate an execution context and to specify what memory addresses are accessible by each execution context. Next, HardScope hardware enforces memory accesses according to the configured rules or access control policies. HardScope is designed and prototyped on top of the RISC-V ISA for embedded applications. More details on HardScope are provided in Appendix G.

The widespread adoption of runtime defenses by major hardware and software vendors, e.g. Intel CET [26], ARM Pointer Authentication (PA) [165], Date Execution Prevention (DEP), Microsoft Microsoft Control Flow Guard (CFG) and ASLR in Linux, MacOS and Windows systems, has driven the evolution of more sophisticated attacks against the different protection schemes: i) ASLR [168, 169, 170, 171, 18], ii) stack canaries [171, 172], iii) pointer integrity solutions [173] including attacks on ARM PA,[1] or iv) CFI [174, 175] including attacks on Intel CET and Microsoft CFG.[2]

3.3 RUNTIME ATTESTATION

In addition to the aforementioned runtime defenses, researchers have recently proposed detection-based solutions of runtime attacks through attestation. Attestation is one of the key approaches for platform integrity that is widely adopted to verify the trustworthiness of software components and detect malware attacks. Remote attestation is a challenge-response protocol that enables the *verifier* to verify the trustworthiness of the *prover*

1 https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html

2 https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf

by comparing the prover's response, *attestation report*, to a reference value. In its basic form, *static attestation*, the attestation report is computed over a fresh challenge sent by the verifier and the memory content of the prover using a cryptographic hash function and a secret key shared with the verifier. This implies the presence of a *trust anchor* on the prover in order to compute the attestation report. This trust anchor could be as minimal as a secret key and a piece of immutable code that reads the memory content and computes a proof of integrity [47, 176]. Nevertheless, static attestation provides no information on software behavior at runtime.

To enable the detection of runtime attacks, the execution of a program is tracked, measured in an attestation report and sent to the verifier. To verify the execution integrity the measured execution is compared to a reference execution model. The two currently known runtime attestation schemes are Control-Flow Attestation (CFA) and Data-Flow Attestation (DFA).

Reference Model. For CFA a reference execution model is typically based on the program's CFG, where the nodes of the graph represent basic blocks and the edges represent valid control-flow transfers/transitions. A basic block consists of a sequence of program instructions that ends with one of the control-transfer instructions, such as jump, branch or return instructions. Each edge in the graph is represented as a tuple (source, destination) using for example the memory address of the control-transfer instruction as the source and the memory address of the first instruction in targeted basic block as the destination. The security achieved by comparing to a CFG-based reference execution model is dependent on the constructed CFG [174, 175, 177]. Further, a reference execution model may include data-dependent execution information, e.g. for a given input range of the program what the expected execution paths, i.e. a series of control-flow transitions within the CFG, or which values its loop counters might be. Such information can be acquired by performing dynamic analysis of the code, e.g. symbolic execution. Similarly, for DFA the reference execution model is based on the program's DFG that defines how variables are assigned or used. The verifier is assumed to able to generate the required reference execution model and to have enough computing resources to perform the verification.

3.3.1 *Control-Flow Attestation (CFA)*

We present next state-of-the-art CFA schemes and categorize them based on their underlying trust anchors into TEE-assisted [52, 178], hardware-based [4, 5, 6], hybrid [179, 180] and software-only CFA schemes [181, 182]. Then, we present CFA schemes that tackle the Time of Check Time of Use (TOCTOU) problem and CFA schemes for real-time applications.

TEE-assisted CFA. The first seminal work on CFA was C-FLAT [52]. C-FLAT aims at detecting control-flow and non-control data attacks in embedded systems that support TEE, e.g. ARM TrustZone-M. In C-FLAT the program's binary is instrumented to replace each branch instruction with a trampoline that redirects the execution to a runtime tracer, which identifies the source and destination of the original branch and passes them to a measurement engine. The measurement engine runs in the trusted execution domain and is responsible for computing the hash values. C-FLAT induces a very high performance overhead due to the frequent context switching to the trusted execution domain to compute the hash measurements. Further, its security relies on the integrity of the program binaries and the runtime tracer, therefore, static attestation is a priori.

Hardware-based CFA. To overcome the limitations of C-FLAT, we present LO-FAT [4] that leverages hardware to trace, encode and measure detailed execution information. Although LO-FAT components are tightly integrated within the processor's pipeline, it does not interfere with its execution. LO-FAT uses two signals, the program counter and the executed instruction, from the processor pipeline to identify source-destination pair of control-transfer transitions as well as loop's entry and exit points. These are used to efficiently and uniquely encode the different execution paths. The encoded execution paths and other execution details such as loop iterations are used as metadata. LO-FAT computes a single hash measurement for all executed paths using the source-destination pairs. The computed hash value is then enclosed in an attestation report along with the metadata. The hash value and the metadata are used by the verifier to recover the execution traces and enable the detection of control-flow and non-control data attacks. LO-FAT targets bare-metal embedded applications and requires neither modifications to the program code nor additional software components. That is, it completely relies on trusted hardware components to capture the control-flow behavior, however, LO-FAT still relies on static attestation to ensure the integrity of the binaries prior to its execution. More details on LO-FAT are provided in Appendix D.

Hybrid and Software-only CFA. Hybrid CFA schemes that target embedded systems have been proposed. LAPE [179] targets bare-metal legacy firmware on IoT devices, which is recompiled to generate two components, a trusted part, whose security is protected by a memory protection unit to periodically measure the control-flow transitions of the untrusted part, i.e. untrusted firmware functionalities. TinyCFA [180] is a hybrid CFA scheme for low-end embedded applications running on micro-controllers that extends on the VRASED architecture [176] for static attestation. On the other hand, ScaRR [181] and ReCFA [182] are software-only CFA schemes that rely on the trustworthiness of the kernel to perform the control-flow measurements of high-end user-space programs.

CFA Resilient to Time of Check Time of Use (TOCTOU). TOCTOU attacks is one major concern for remote attestation schemes. TOCTOU attacks exploit the time gap between the measurement time of a victim program and its execution to replace the victim binaries with a malware or a malicious code. In [5], we demonstrate that attestation schemes that follow the concept of *attest-then-run* are vulnerable to TOCTOU attacks. For example, a TOCTOU attack can be performed by an attacker that has a physical access to the victim device's off-chip memory only and can manipulate its contents. Note that CFA schemes can also be vulnerable to TOCTOU attacks, when the injected malicious code's CFG is reformed to match the victim's CFG [5]. To mitigate TOCTOU attacks, we propose the concept of *run-and-attest* and build ATRIUM, the first runtime attestation scheme that captures both the control-flow behavior of the program as well as its executed binaries. This is achieved by leveraging trusted hardware components to capture the executed instructions and control-flow transitions, which are included in the hash measurements. To efficiently handle program' loops while keeping the hardware overhead low, ATRIUM generates several hash measurements rather than a single hash measurement as in LO-FAT. The beginning and end of each hash measurement are defined automatically based on loop entry and exit points. Thus, each hash measurement represents a unique execution path outside or within loops. The attestation report includes all hash measurements and other data-dependent execution information to enable the detection of control-flow and non-control data attacks. More details on ATRIUM are provided in Appendix E.

CFA for Real-Time Applications. To tackle the real-time constraints of some embedded applications we propose CHASE [6], a *hardware-based* runtime attestation scheme that operates in four modes. CHASE's modular design allows it to be configured dynamically at runtime to operate in one of the available modes depending on the current security requirements and the tolerable performance overhead at the time of attestation. The four modes of attestation, from the lowest performance to the highest performance overhead, are: i) on-device CFA, ii) on-device CFI enforcement, iii) remote CFA, iv) remote runtime attestation to detect TOCTOU attacks. On-device CFA mode guarantees low-latency detection of illegal control-flow transfers while incurring minimal performance overhead, thus making it suitable for real-time applications. Local verification is performed by the trust anchor with the help of the integrity-protected program's CFG that is uploaded on the device. At runtime, CFG edges, i.e. source-destination pairs, are compared to the source-destination of the currently executing control-flow transfer to verify it. CHASE is presented in Appendix F.

DIAT [178] is a CFA scheme for collaborative autonomous systems with safety critical applications. In DIAT the generation and the verification of CFA reports are carried on by peer constrained embedded devices within TEEs such as ARM TrustZone-M. Therefore, rather than measuring and verifying long execution paths, the measurement and the verification processes in DIAT are restricted to measuring and verifying control-flow transitions within the CFG regardless of their orders. This is achieved by leveraging multiset hash functions [183]. Nevertheless, the induced performance overhead is still significant for constrained real-time applications.

3.3.2 *Data-Flow Attestation (DFA)*

Beyond CFA, LiteHAX [184] enables the detection of both control-flow and data-oriented attacks for bare-metal embedded applications. LiteHAX captures both the control-flow and data-flow transitions of a program to enable a remote verifier to detect illegal memory accesses and control-flow transfers. While LiteHAX relies on signal values extracted directly from the processor's pipeline, the runtime attestation scheme proposed in [185] captures the control-flow and data-flow transitions by monitoring the address and data signals on the system bus and relies on a runtime integrity model stored on the device to identify the instructions and their addresses.

4 TRUST IN CLOUD FPGAS

My Contributions. This chapter is based on the results of two papers published in the IEEE European Symposium on Security and Privacy (EuroS&P) and the International Symposium On Field-Programmable Custom Computing Machines (FCCM):

[8] Ghada Dessouky, Ahmad-Reza Sadeghi, Shaza Zeitouni. SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities. In *Proceedings of the 6th IEEE European Symposium on Security and Privacy* (EuroS&P'21), 2021.

I led this work and contributed with the co-authors to the discussions on the deployment model, stakeholders, attack landscape and drawing the analogies to the trusted computing environments on CPUs. Ghada Dessouky further focused on the discussions of microarchitectural attacks. While I focused on the classification and discussions of the different remote physical attacks and their state-of-the-art defenses, their counterparts in the general-purpose computing paradigm, and the discussions of the trusted computing base on cloud FPGAs.

[9] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, Nele Mentens. Trusted Configuration in Cloud FPGAs. In *Proceedings of the 29th IEEE International Symposium On Field-Programmable Custom Computing Machines* (FCCM'21), 2021.

I led this work and contributed to the discussions of the core idea with the co-authors. I focused on the development of the protocol and the security analysis of the protocol. Further I worked on the design, the implementation and the evaluation of the trust anchor on the FPGA. Jo Vliegen contributed to the discussions of the core idea with the co-authors and assisted in developing the protocol. Tommaso Frassetto focused on the implementation and evaluation of the virus scanner in Intel SGX.

Chapter Outline. In the following, we present an overview on FPGA security including IP protection before their deployment in datacenters in Section 4.1. Then, we discuss FPGA deployment models in the cloud, the emerging security challenges and solutions including our work in Section 4.2.

4.1 FPGA SECURITY

FPGAs are reconfigurable ICs that can be electrically (re)programmed by end users using binary files, the *bitstreams*, to implement different digital circuits of their own choice. FPGAs come in different computational capabilities and can operate as fully-fledged stand-alone computing systems or be integrated as hardware accelerators in more complex systems, which make them suitable for the deployment in different computing paradigms, IoT, edge or cloud. To configure the FPGA with the intended circuit, the configuration engine, which resides on the FPGA chip, reads the encoded configuration information within the bitstream and uses it to write the configuration memory of the FPGA accordingly. FPGA configuration memory refers to the memory that holds the configuration patterns of a given circuit. Existing FPGAs leverage different memory technologies for their configuration memories, such SRAM, flash or anti-fuse. We focus on SRAM-based FPGAs since SRAM-based FPGAs are the most widespread FPGAs and are the ones deployed currently in the cloud.

Attacks. In general, physical attacks that can be conducted on ICs are also applicable to FPGAs. However, the flexible nature of FPGAs introduces a new attack surface, when deployed in-field in an untrusted environment. For example, on SRAM-based FPGAs, i.e. whose configuration memory is made of SRAM cells, configuration data is lost at power-off and thus an off-chip non-volatile memory is required to store the bitstream until it is loaded again on the FPGA after the next power-up. This would allow an attacker to i) read out the existing bitstream to reverse-engineer the original design or copy it to a similar FPGA without authorization or ii) overwrite the current bitstream with an unauthorized (malicious) bitstream. An overview of FPGA-specific attacks are presented in [186].

Intellectual Property (IP) Protection. Therefore, to prevent such attacks, the bitstream must be provisioned securely between power cycles. Thereby, a trust anchor on the FPGA is required to enable the configuration of authorized bitstreams and to prevent the read out of proprietary designs. Being a topic of concern in the early era of reconfigurable ICs, several works have been conducted in this direction [187, 85, 188, 189, 190, 191]. Further, FPGA vendors have reacted already to these requirements, as evident in more recent FPGAs [37, 38]. Recent FPGAs are equipped with hard-coded cryptographic engines that enable the verification and decryption of users' bitstreams prior to their configuration on the FPGA. However, the user, i.e. the owner of the FPGA, is responsible for enabling the protection of bitstreams on the FPGA. That is, the user must provision the keys on the FPGA's trust anchor in a secure environment before it is deployed in-field.

4.2 FPGAs IN THE CLOUD

In recent years, FPGAs have made their way into datacenters along with GPUs and ASICs as hardware accelerators of compute-intensive services and applications. Unlike ASICs, whose functionalities cannot be changed after fabrication, FPGAs and GPUs are flexible in the sense that the programmed functionality can be updated/modified when required, yet FPGAs provide higher flexibility, by supporting various data types, and higher energy efficiency compared to general-purpose computing platforms [192, 193]. Therefore, they have been perceived to bring the best of both hardware and software worlds.

4.2.1 *Deployment Models*

FPGAs are deployed in the cloud under two different deployment models: Acceleration-as-a-Service (AaaS) or FPGA-as-a-Service (FaaS). Both models provide *acceleration* for a wide range of compute-intensive workloads, such as Machine Learning (ML), genomic data processing, and other scientific computations.

AaaS Model. In AaaS, FPGAs are dedicated to accelerate specific tasks, e.g. web search and network encryption, that are pre-defined by the Cloud Service Provider (CSP). In this model, clients have no access to FPGA fabric, but can benefit from the accelerated services. Microsoft Azure was among the first datacenters to introduce FPGAs in cloud computing by augmenting their processors with an interconnected and configurable compute layer of FPGAs to accelerate the Bing web search [194] and network encryption [195]. More recently, Microsoft launched Brainwave project for FPGA-accelerated real-time ML inference for Bing web search and Skype language translations [196]. To this end, the FPGA fabric in the AaaS model is not directly accessible by the clients, yet FPGAs are part of the underlying infrastructure of the cloud. Thus, trust assumptions and requirements by the CSPs would include FPGA vendors and potential IP providers, if external proprietary designs will be configured on the FPGAs.

FaaS Model. FaaS, on the other hand, enables a more flexible usage model where clients can directly configure their allocated FPGAs with the desired functionality. In fact, academic proposals suggested to further allocate a single FPGA to multiple clients/tenants to maximize FPGA utilization [197, 198, 199, 200, 201, 202]. This is achieved by leveraging the partial reconfigurability of FPGAs, in which a single FPGA can be split into logically-isolated regions and each region can be reconfigured at runtime without disturbing other regions. This sharing model is known as spatial multi-tenancy. On the other hand, prominent CSPs, e.g. Amazon [55], Huawei [203], and Alibaba [56], still offer their clients dedicated FPGAs. This sharing model is known as temporal multi-tenancy, i.e. clients can only use the same FPGA after it is released. Therefore, the FaaS model introduces additional security challenges that arise due to the flexible nature of FPGAs.

In [8], we present a systematization of knowledge on the security of multi-tenant cloud FPGAs. We focus on the FaaS model and present a through analysis on the different stakeholders, threats, trust assumptions and security requirements. Concurrent to our work, trust assumptions and threat models within a remote FPGA deployment model in general are discussed in [204].

In the following, we focus on FPGA security challenges that stem from the recent deployment model of FPGAs in the cloud, the FaaS.

4.2.2 *Attacks on Cloud FPGAs*

The FPGA's configurable fabric and the flexibility offered to FPGA programmers to precisely control the underlying components allow the realization of various digital circuits including voltage and temperature sensors that work by measuring propagation delay differences in a given circuit to infer changes in power consumption or temperature on the FPGA [53, 54]. These tiny voltage sensors have even proven their effectiveness by measuring power consumption on neighboring devices that share the same power supply [57]. Given the fact that in the cloud environment the FPGAs can share the power supply system with other components in the infrastructure and the clients are allowed to configure the FPGAs freely, various *remote* physical attacks have been demonstrated in spatial and temporal multi-tenancy settings. These attacks mainly deploy two types of *malicious primitives*, voltage/temperature sensors or power viruses, which can draw an excessive amount of current from the power supply of the FPGA.

In [8] we distinguish between two classes of attacks with respect to FPGAs. The first one includes well-known attacks in typical CPU-based cloud computing settings, e.g. Rowhammer and cache side-channel attacks, and does not require the configuration of malicious primitives on cloud FPGAs. In this direction we shed the lights on the various lessons from the CPU-based computing and draw analogies to FPGA-based computing in order to have better understanding of the security challenges and their potential treatments on FPGA-based computing in the cloud. The second class of attacks includes the different variants of remote physical attacks, such as fault-injection and power/thermal side-channel attacks on co-clients as well as DoS attacks on the cloud infrastructure, and can be conducted by configuring malicious primitives on the FPGA. We analyze and categorize the different attacks in temporal and spatial multi-tenancy settings and further scrutinize the proposed defenses against these attacks. More details are provided in Appendix H.

4.2.3 *Defenses on Cloud FPGAs*

Mitigating Remote Physical Attacks. We classify state-of-the-art defenses against remote physical attacks into static proactive solutions that mitigate the threats before configuration on the FPGA and runtime solutions that aim to address the gap that static approaches cannot comprehensively close. Nevertheless, runtime solutions try to address different forms of attacks, e.g. power side-channel attacks, thermal side-channel attacks or fault injection attacks, individually. Moreover, some of these defenses are dependent on the circuit to be protected and thus should be implemented by the clients [205]. Consequently, runtime defenses incur higher overhead and have limited effectiveness compared to proactive countermeasures. Among the most promising proactive countermeasures that would hinder most of the currently-known remote physical attacks are virus scanners. They allow the CSP to vet the FPGA's bitstream in order to detect the presence of malicious circuits, i.e. voltage and temperature sensors and power viruses, prior to their deployment on cloud FPGAs [206, 207]. Nevertheless, these virus scanners require access to clients' bitstreams.

Remote IP Protection. Clients' bitstreams may include proprietary designs, yet bitstream encryption is not supported on cloud FPGAs. On the one hand, clients have no physical access to the FPGA and there is no direct and secure means to program the user's secret key into the FPGA's trust anchor. On the other hand, multiple clients might be scheduled to use the same FPGA whether within the temporal or the spatial multi-tenancy models. Therefore, direct utilization of the existing hard-coded cryptographic engines on commodity FPGAs by clients for IP protection on the cloud is not feasible for the two reasons: i) secure remote key provisioning step is not supported and ii) the key provisioning scheme is not designed to support multiple clients.

To solve IP protection issue on cloud FPGAs, academic initiatives proposed the use of an initial bitstream to configure a trust anchor on cloud FPGAs for clients' bitstreams protection [208, 209]. The initial bitstream contains different cryptographic cores for decryption of clients' bitstreams and for secret key exchange or generation. Therefore, the initial bitstream itself must be protected. This is achieved through FPGA vendor support, either by programming the secret key on the FPGA trust anchor [208] or by configuring the initial bitstream on the FPGA [209] before deployment in the cloud. The latter assumes FPGA to be continuously powered, even during shipping to the cloud, to maintain its configuration. Nevertheless, having IP protection enabled on cloud FPGAs, malicious primitives, which can be contained within clients encrypted bitstreams, would be overlooked.

4.2.4 *Mutual Trust on Cloud FPGAs*

To summarize the current mutual trust problem: clients require to protect their proprietary designs by encrypting their bitstreams, which is not allowed on cloud FPGAs, while access to FPGA configurations is required to inspect for malicious circuits prior to configuration on cloud FPGAs. To solve this paradoxical problem of client's IP protection and prevention of malicious FPGA configurations, we propose the first scheme to satisfy the requirements of clients and CSPs in [9].

Our solution comprises a cryptographic protocol by the two parties the client and the CSP and requires the existence of a TEE and a trust anchor on the FPGA that can support cloud deployment models. To vet a client's bitstream prior to configuration on a cloud FPGA, a design check on the bitstream is executed inside a TEE and a proof of execution including a status report is provided to the CSP. We examine two options where the TEE resides, on the client side or on the CSP side, and show the trade-offs between the two options. Then, the CSP verifies the status report and allows the encrypted bitstream to be loaded on the FPGA, where it is verified and decrypted on-the-fly with the help of a trust anchor of our design that resides on the FPGA. As such, the CSP has no access to the decrypted bitstream. We further explore the design options of the trust anchor being either configurable or hard-coded on the FPGA fabric and propose a remote secret key provisioning and management scheme that can support multiple clients acquiring a single FPGA instance by utilizing PUFs and public-key cryptography.

Our solution requires the support of hardware vendors, TEE and FPGA vendors, which is a reasonable assumption in the hardware-assisted trusted computing paradigm. Finally, we prototype our configurable trust anchor on the FPGA and show the feasibility of our scheme on commodity FPGAs. By solving both security challenges, we provide the means for practical isolated/trusted execution on cloud FPGAs and thus enabling secure processing of sensitive data on cloud FPGAs. We provide more details on our solution and the different design options in Appendix I.

5 CONCLUSION

Hardware-assisted security aims at protecting computing systems against various software-based attacks including malware and runtime attacks. Hardware-assisted security architectures imply the existence of a trust anchor in the hardware of a computing system to support various cryptographic protocols and security features. During the last two decades, we have witnessed great advances in this direction with the deployment of various hardware-based security solutions into today's computing systems such as Trusted Platform Module, Physically Unclonable Functions (PUFs), ARM TrustZone, Intel Software Guard Extension and Intel Control-Flow Enforcement Technology.

The main focus of this dissertation is the design of trust anchors for hardware-assisted security architectures. First, we address PUFs, one of the basic primitives of a trust anchor, which are used to generate device-specific identifiers and secret keys. Then, we consider the design of trust anchors to support different security features, namely, runtime protection and trusted configuration. In the following, we briefly recap on the main contributions of this thesis in Section 5.1 and present future research directions in Section 5.2.

5.1 DISSERTATION SUMMARY

5.1.1 *Security Evaluation of Physically Unclonable Functions*

In Chapter 2 we presented an overview on the current research work on PUFs and demonstrated our attacks on the SRAM PUF, the Rowhammer PUF and a set of hybrid memristor-based PUFs.

We performed a physical fault injection attack on SRAM PUFs by precisely controlling the voltage supply to the device embedding the PUF in order to induce the remanence decay effect and gradually allow the SRAM cells revert to their start-up values. We showed then how to recover the original PUF response with the help of faulty PUF responses and differential fault analysis. The attack exploits the fact that in constrained devices SRAM memory, which is used to extract the PUF response at boot up, is accessible by other processes at runtime and hence are accessible to the attacker.

Then, we performed two attacks on a variant of the runtime-accessible decay-based DRAM PUFs, the Rowhammer PUF. In the first attack we demonstrated how PUF reliability is deteriorated by accessing only neighboring locations to the PUF region thus leading to faulty PUF responses and keys. In the second attack, we performed a modeling attack on the Rowhammer PUF using standard interpolation algorithms. Our results indicate that PUF responses can be predicted and the Rowhammer PUF is not secure for deployment in light-weight authentication protocols as suggested by the authors.

Finally, we showed that even memristor-based PUFs, which promised higher reliability and unpredictability over CMOS-only PUFs, are prone to modeling attacks. We reproduced a set of memristor-based PUFs and tested them using a set of Machine Learning (ML) algorithms. Our results indicate that the inspected PUFs and their XOR-based versions are vulnerable to modeling attacks.

5.1.2 *Hardware-assisted Runtime Protection*

In Chapter 3 we gave an overview on the different runtime attacks and defense strategies. We presented then our novel hardware-assisted runtime enforcement and attestation schemes that target bare-metal embedded applications and rely on their hardware trust anchors to achieve the required security guarantees.

HardScope is a novel ISA extension based on RISC-V to mitigate the currently-known data-oriented attacks. HardScope trust anchor provides fine-grained context-specific memory isolation by enforcing access control on function variables in the memory based on policies defined at compile-time.

LO-FAT trust anchor captures, encodes and reports the fine-grained control-flow behavior of the attested program to the verifier for the detection of control-flow and non-control data attacks. LO-FAT requires neither modifications to the program code nor additional software components. However, LO-FAT still relies on static attestation to ensure the integrity of the binaries prior to its execution.

To mitigate Time of Check Time of Use (TOCTOU) attacks on attestation schemes, we proposed the concept of run-and-attest in ATRIUM, the first runtime attestation scheme that captures both the control-flow behavior of the program as well as its executed binaries in a single attestation report. ATRIUM enables the detection of TOCTOU as well as control-flow and non-control data attacks.

Finally, we presented CHASE, a runtime attestation scheme for embedded applications with real-time constraints. CHASE on-device Control-Flow Attestation mode allows its trust anchor to capture and locally-verify control-flow transfers on-device using program's Control-Flow Graph. This mode guarantees low-latency detection of illegal control-flow transfers while incurring minimal performance overhead.

5.1.3 *Hardware-assisted Trusted Configuration*

In Chapter 4 we explored the current trend of leveraging FPGAs in the cloud for acceleration of cloud services and clients' applications.

We investigated the security challenges and opportunities of cloud FPGAs in the FPGA-as-a-Service (FaaS) deployment model. Given the fact that in FaaS model clients are allowed to configure the FPGAs freely, various remote physical attacks have been demonstrated. We analyzed and categorized the different attacks in FaaS model and scrutinized the proposed defenses against these attacks. Further, we drew attention to the mutual trust problem of cloud FPGAs: clients require to protect their proprietary designs, while Cloud Service Providers (CSPs) do not support the use of encrypted FPGA configurations. In fact, CSPs require access to FPGA configurations to inspect for malicious circuits that can be used to launch physical attacks remotely, prior to configuration.

Then we tackled this problem by presenting the first trusted configuration scheme for cloud FPGAs. Our scheme provides the CSP with an authentic proof that the encrypted bitstream has been vetted for malicious primitives with the help of a Trusted Execution Environment (TEE). If admitted, the encrypted bitstream is forwarded to our trust anchor, which provides a secret key provisioning and management scheme that supports multiple clients, for decryption and configuration on the FPGA. By solving both security challenges, we provide the means for practical isolated/trusted execution on cloud FPGAs and thus enabling secure processing of sensitive data on cloud FPGAs.

5.2 FUTURE RESEARCH DIRECTIONS

5.2.1 *Physically Unclonable Functions*

Scrutinizing PUF Primitives. In recent years, prominent progress has been made in device technologies beyond CMOS, such as carbon nanotube field-effect transistors, spintronic-logic devices, fully depleted silicon on insulator and many others. Consequently, several PUF designs that leverage these emerging technologies have been proposed, yet most of the proposed PUFs are only demonstrated in simulation environments and their security properties are not verified. Thus, there is a need to have a unified framework for evaluating the claimed security properties of recent hybrid PUF designs in unified/comparable simulation environments and operating conditions for fair comparison. This will provide a deep insight on the different capabilities of these technologies and their suitability for the design of novel modeling-resilient PUFs.

On the other hand, PUFs have made their way into commercial products, such as Intel Stratix 10 SoCs [210] and Microsemi SmartFusion2 SoCs [211] for the purpose of generating device-specific keys. Another interesting research direction is to investigate the security of these real-world PUF implementations.

5.2.2 *Runtime Attestation*

Software-Hardware Co-Design. In our hardware-based attestation schemes we aimed to keep the performance overhead minimal, however, this comes at the cost of higher area overhead, in terms of memory and logic. Our trust anchors deploy different components to track the execution, encode control-flow transfers and hash them. For example, one basic hardware component is responsible filtering out instructions and memory addresses to identify loop entry and exist points. However, it can only handle a limited number of loops and nested loops simultaneously, which can be different from program to another. Supporting more (nested) loops and recursive functions would complicate the design process and increase the hardware complexity. One option to tackle this issue is by exploring the software-hardware co-design space to build an efficient scheme that would strike a balance between flexibility, area and performance overhead.

Attestation & Verification of Complex Software. Further, our attestation schemes support bare-metal embedded programs only. Consequently, extending the scope of our attestation schemes to support embedded programs running on top of operating systems will allow the coverage of more application areas. In this direction, it is also interesting to explore the challenges and opportunities for hardware-assisted runtime attestation of complex user-space applications on desktop systems. The goal of such scheme is two-fold: reduce the performance overhead, compared to software-only schemes [181, 182], by offloading intensive computations to hardware components while minimizing the trust assumptions on software components. On the other hand, verifying attestation reports of complex software is expected to be challenging due to the significantly large space of possible execution paths. Thus, efficient verification methods should be also investigated. In this direction ML capabilities for the verification of runtime behavior of complex software can be explored.

5.2.3 *Trust in Cloud FPGAs*

Practical Trusted FPGA-based Cloud Computing. In our scheme for trusted FPGA configuration [9] the client's bitstream is encrypted using an FPGA-specific secret key. In order for the client to use different cloud FPGAs, the client must re-encrypt the bitstream with different secret keys. This further calls for reinspecting the FPGA configuration. One feasible approach to tackle this problem is by building a database of the previously scanned clients' designs and store their corresponding hash values and scan reports. Before inspecting a client configuration a search for that FPGA configuration in the database is performed using its hash value. Nevertheless, this solution still implies the initialization and running of a TEE enclave to perform the search. Therefore, alternatives such as different encryption schemes that allow the encryption with a single key and the decryption with different keys [212] can be investigated. This would enable a more practical and cost effective approach for trusted cloud FPGA configuration.

Trusted Cloud FPGAs Applications. Establishing trusted configuration on cloud FPGAs is the first step towards trusted execution on these platforms. The next step is to leverage the computational power of FPGA TEEs to enable efficient and secure processing of compute-intensive real-world applications. An example of such workloads is federated ML, where aggregation of federated ML models is to be performed in an untrusted environment. Thus, the federated ML models are prone to powerful inference attacks by an untrusted aggregator. Compared to existing solutions of outsourcing Secure Multi-Party Computation [63] in a semi-honest model, leveraging FPGA TEEs will reduce the computation and communication overhead significantly. In this direction, it is also interesting to investigate the deployment of defenses against malicious clients that aim to sabotage the aggregated model by injecting poisoned models during the aggregation process [213]. Thus, integrating a poisoning defense in the aggregation process will protect the confidentiality of clients' models and the integrity of the aggregated model.

BIBLIOGRAPHY

[1] Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, and Ahmad-Reza Sadeghi. Remanence decay side-channel: The PUF case. *IEEE Transactions on Information Forensics and Security (TIFS)*, 11(6), 2015.

[2] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. It's hammer time: how to attack (rowhammer-based) DRAM-PUFs. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*. ACM, 2018.

[3] Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, and Ahmad-Reza Sadeghi. On the security of strong memristor-based physically unclonable functions. In *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

[4] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N Asokan, and Ahmad-Reza Sadeghi. LO-FAT: Low-overhead control flow attestation in hardware. In *Design Automation Conference (DAC)*. ACM, 2017.

[5] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. ATRIUM: Runtime attestation resilient under memory attacks. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017.

[6] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, and Ahmad-Reza Sadeghi. CHASE: A configurable hardware-assisted security extension for real-time systems. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019.

[7] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N Asokan, and Ahmad-Reza Sadeghi. HardScope: Hardening embedded systems against data-oriented attacks. In *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019.

[8] Ghada Dessouky, Ahmad-Reza Sadeghi, and Shaza Zeitouni. SoK: Secure fpga multi-tenancy in the cloud: Challenges and opportunities. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021.

[9] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, and Nele Mentens. Trusted configuration in cloud fpgas. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021.

[10] Geoff McDonald, Liam O Murchu, Stephen Doherty, and Eric Chien. Stuxnet 0.5: The missing link. *Symantec Report*, 2013.

[11] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *USENIX Security Symposium*. USENIX Association, 2017.

[12] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and analysis of hajime, a peer-to-peer iot botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[13] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Ryan Court, Kevin Snow, Fabian Monrose, and Manos Antonakakis. The circle of life: A Large-Scale study of the IoT malware lifecycle. In *USENIX Security Symposium*. USENIX Association, 2021.

[14] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the expressiveness of return-into-libc attacks. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*. Springer Berlin Heidelberg, 2011.

[15] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1), 2012.

[16] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2011.

[17] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*. USENIX Association, 2005.

[18] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[19] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3), 2014.

[20] Yinqian Zhang. Cache side channels: State of the art and research opportunities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[22] Trusted Computing Group. Trused platform module (TPM) 2.0: A brief introduction, 2019. URL https://trustedcomputinggroup.org/work-groups/trusted-platform-module.

[23] Intrinsic ID. URL https://www.intrinsic-id.com/physical-unclonable-function.

[24] ARM Ltd. Security technology: building a secure system using TrustZone technology, 2008. URL http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC009492C_trustzone_security_whitepaper.pdf.

[25] Intel. Intel software guard extensions programming reference, 2014. URL https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf.

[26] Intel. A technical look at Intel's Control-flow Enforcement Technology, 2017.

[27] Kees Cook. Kernel address space layout randomization. *Linux Security Summit*, 2013.

[28] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[29] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1), 2017.

[30] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006.

[31] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *Network and Distributed Systems Security (NDSS) Symposium*, 2016.

[32] AMD. Amd sev-snp. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf, 2019.

[33] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*. USENIX Association, 2016.

[34] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[35] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *The European Conference on Computer Systems (EuroSys)*. ACM, 2020.

[36] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with CUstomizable and resilient enclaves. In *USENIX Security Symposium*. USENIX Association, 2021.

[37] *Using encryption and authentication to secure an UltraScale/UltraScale+ FPGA bitstream*. AMD Xilinx Inc., 10 2018. Rev. 1.3.

[38] Intel. Intel stratix 10 configuration user guide. https://www.intel.com/content/www/us/en/docs/programmable/683762/22-1/secure-device-manager.html, 2022.

[39] Jae W Lee, Daihyun Lim, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *Symposium on VLSI Circuits. Digest of Technical Papers (IEEE Cat. No. 04CH37525)*. IEEE, 2004.

[40] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2007.

[41] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Lightweight secure pufs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2008.

[42] Daniel E Holcomb, Wayne P Burleson, and Kevin Fu. Power-up sram state as an identifying fingerprint and source of true random numbers. *IEEE Transactions on Computers (TC)*, 58(9), 2009.

[43] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. Low-overhead implementation of a soft decision helper data algorithm for sram pufs. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2009.

[44] Fatemeh Tehranipoor, Nima Karimian, Kan Xiao, and John Chandy. Dram based intrinsic physical unclonable functions for system level security. In *Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2015.

[45] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Intrinsic rowhammer pufs: Leveraging the rowhammer effect for improved security. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017.

[46] Wenjie Xiong, André Schaller, Nikolaos A. Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Run-time accessible dram pufs in commodity devices. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2016.

[47] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Network and Distributed Systems Security (NDSS) Symposium*, 2012.

[48] PaX Team. Pax address space layout randomization (aslr). http://pax.grsecurity.net/docs/aslr.txt.

[49] Microsoft. Data execution prevention, 2006. URL https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[50] Martın Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Cfi: Principles, implementations, and applications. In *ACM Conference and Computer and Communications Security (CCS)*. ACM, 2005.

[51] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[52] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-flow attestation for embedded systems software. In *ACM SIGSAC Conference on Computer and Communications Security (CSS)*. ACM, 2016.

[53] Kenneth M Zick and John P Hayes. Low-cost sensing with ring oscillator arrays for healthier reconfigurable systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(1), 2012.

[54] Dennis RE Gnad, Fabian Oboril, Saman Kiamehr, and Mehdi B Tahoori. Analysis of transient voltage fluctuations in FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016.

[55] Amazon AWS. Amazon EC2 F1. https://aws.amazon.com/ec2/instance-types/f1/.

[56] Alibaba Cloud. FPGA-based Compute-Optimized Instance Families. https://www.alibabacloud.com/help/doc-detail/108504.htm, 2019.

[57] Ilias Giechaskiel, Kasper Bonne Rasmussen, and Jakub Szefer. C<sup>3</sup>apsule: Cross-fpga covert-channel attacks through power supply unit leakage. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[58] Tuan La, Khoa Pham, Joseph Powell, and Dirk Koch. Denial-of-service on fpga-based cloud infrastructure-attack and defense. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021.

[59] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[60] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. Automated synthesis of optimized circuits for secure computation. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[61] Ebrahim M. Songhori, Thomas Schneider, Shaza Zeitouni, Ahmad-Reza Sadeghi, Ghada Dessouky, and Farinaz Koushanfar. GarbledCPU: A mips processor for secure computation in hardware. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016.

[62] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *Network and Distributed System Security (NDSS) Symposium*, 2017.

[63] H. Fereidooni, S. Marchal, M. Miettinen, A. Mirhoseini, H. Mollering, T. Nguyen, P. Rieger, A. Sadeghi, T. Schneider, H. Yalame, and S. Zeitouni. Safelearn: Secure aggregation for private federated learning. In *IEEE Security and Privacy Workshops (SPW)*. IEEE, 2021.

[64] Johannes Buchmann, Ghada Dessouky, Tommaso Frassetto, Ágnes Kiss, Ahmad-Reza Sadeghi, Thomas Schneider, Giulia Traverso, and Shaza Zeitouni. Safe: A secure and efficient long-term distributed storage system. In *International Workshop on Security in Blockchain and Cloud Computing*. ACM, 2020.

[65] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Shaza Zeitouni. Seed: Secure non-interactive attestation for embedded devices. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*. ACM, 2017.

[66] Yossef Oren, Ahmad-Reza Sadeghi, and Christian Wachsmann. On the effectiveness of the remanence decay side-channel to clone memory-based pufs. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2013.

[67] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. In *Design Automation Conference (DAC)*. ACM, 2003.

[68] Yohei Hori, Takahiro Yoshida, Toshihiro Katashita, and Akashi Satoh. Quantitative and statistical performance evaluation of arbiter physical unclonable functions on fpgas. In *International Conference on Reconfigurable Computing and FPGAs*. IEEE, 2010.

[69] Christina Brzuska, Marc Fischlin, Heike Schröder, and Stefan Katzenbeisser. Physically uncloneable functions in the universal composition framework. In *Advances in Cryptology (CRYPTO)*. Springer Berlin Heidelberg, 2011.

[70] Abhranil Maiti, Vikash Gunreddy, and Patrick Schaumont. A systematic method to evaluate and compare the performance of physical unclonable functions. In *Embedded Systems Design with FPGAs*. Springer New York, 2013.

[71] Frederik Armknecht, Daisuke Moriyama, Ahmad-Reza Sadeghi, and Moti Yung. Towards a unified security model for physically unclonable functions. In *Topics in Cryptology (CT-RSA)*. Springer International Publishing, 2016.

[72] Ulrich Rührmair, Heike Busch, and Stefan Katzenbeisser. Strong pufs: Models, constructions, and security proofs. In *Towards Hardware-Intrinsic Security: Foundations and Practice*. Springer Berlin Heidelberg, 2010.

[73] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8), 2014.

[74] Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. Secure lightweight entity authentication with strong pufs: Mission impossible? In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2014.

[75] Jeroen Delvaux, Roel Peeters, Dawu Gu, and Ingrid Verbauwhede. A survey on lightweight entity authentication with strong pufs. *ACM Computing Surveys (CSUR)*, 48(2), 2015.

[76] Jeroen Delvaux. Machine-learning attacks on polypufs, ob-pufs, rpufs, lhs-pufs, and puf-fsms. *IEEE Transactions on Information Forensics and Security (TIFS)*, 14(8), 2019.

[77] Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. Helper data algorithms for puf-based key generation: Overview and analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 34(6), 2014.

[78] Jeroen Delvaux, Dawu Gu, Ingrid Verbauwhede, Matthias Hiller, and Meng-Day (Mandel) Yu. Efficient fuzzy extraction of puf-induced secrets: Theory and applications. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2016.

[79] Charles Herder, Ling Ren, Marten Van Dijk, Meng-Day Yu, and Srinivas Devadas. Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 14(1), 2016.

[80] Matthias Hiller, Ludwig Kürzinger, and Georg Sigl. Review of error correction for pufs and evaluation on state-of-the-art fpgas. *Journal of Cryptographic Engineering*, 10(3), 2020.

[81] Roel Maes. *Physically Unclonable Functions: Constructions, Properties and Applications*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2012. URL https://www.esat.kuleuven.be/cosic/publications/thesis-211.pdf.

[82] Aydin Aysu, Ege Gulcan, Daisuke Moriyama, Patrick Schaumont, and Moti Yung. End-to-end design of a puf-based privacy preserving authentication protocol. In

Cryptographic Hardware and Embedded Systems (CHES). Springer Berlin Heidelberg, 2015.

[83] Ulrich Rührmair, Christian Jaeger, Matthias Bator, Martin Stutzmann, Paolo Lugli, and György Csaba. Applications of high-capacity crossbar memories in cryptography. *IEEE Transactions on Nanotechnology (TNANO)*, 10(3), 2011.

[84] Meng-Day Yu, Matthias Hiller, Jeroen Delvaux, Richard Sowell, Srinivas Devadas, and Ingrid Verbauwhede. A lockdown technique to prevent machine learning on pufs for lightweight authentication. *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, 2(3), 2016.

[85] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2007.

[86] Daniel E. Holcomb and Kevin Fu. Bitline puf: Building native challenge-response puf capability into any sram. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2014.

[87] Anys Bacha and Radu Teodorescu. Authenticache: Harnessing cache ecc for system authentication. In *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[88] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2002.

[89] Abhranil Maiti and Patrick Schaumont. Improved ring oscillator puf: An fpga-friendly secure primitive. *Journal of Cryptology*, 24(2), 2011.

[90] Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Side-channel analysis of pufs and fuzzy extractors. In *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer Berlin Heidelberg, 2011.

[91] Dominik Merli, Johann Heyszl, Benedikt Heinz, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Localized electromagnetic analysis of ro pufs. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2013.

[92] Sheng Wei, James B Wendt, Ani Nahapetian, and Miodrag Potkonjak. Reverse engineering and prevention techniques for physical unclonable functions using side channels. In *Design Automation Conference (DAC)*. ACM, 2014.

[93] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Physical unclonable functions and public-key crypto for fpga IP protection. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2007.

[94] Ahmad-Reza Sadeghi, Ivan Visconti, and Christian Wachsmann. Enhancing RFID security and privacy by physically unclonable functions. In *Towards Hardware-Intrinsic Security: Foundations and Practice*. Springer Berlin Heidelberg, 2010.

[95] Patrick Koeberl, Jiangtao Li, Anand Rajan, Claire Vishik, and Wei Wu. A practical device authentication scheme using SRAM PUFs. In *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer Berlin Heidelberg, 2011.

[96] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Annual International Cryptology Conference (CYPTO)*. Springer Berlin Heidelberg, 1997.

[97] Dmitry Nedospasov, Jean-Pierre Seifert, Clemens Helfmeier, and Christian Boit. Invasive puf analysis. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2013.

[98] Clemens Helfmeier, Christian Boit, Dmitry Nedospasov, and Jean-Pierre Seifert. Cloning physically unclonable functions. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2013.

[99] Heiko Lohrke, Shahin Tajik, Christian Boit, and Jean-Pierre Seifert. No place to hide: Contactless probing of secret data on fpgas. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2016.

[100] Shahin Tajik, Enrico Dietz, Sven Frohmann, Jean-Pierre Seifert, Dmitry Nedospasov, Clemens Helfmeier, Christian Boit, and Helmar Dittrich. Physical characterization of arbiter pufs. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2014.

[101] Shahin Tajik, Enrico Dietz, Sven Frohmann, Helmar Dittrich, Dmitry Nedospasov, Clemens Helfmeier, Jean-Pierre Seifert, Christian Boit, and Heinz-Wilhelm Hübers. Photonic side-channel analysis of arbiter pufs. *Journal of Cryptology*, 30(2), 2017.

[102] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. Modeling attacks on physical unclonable functions. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010.

[103] Ulrich Rührmair, Jan Sölter, Frank Sehnke, Xiaolin Xu, Ahmed Mahmoud, Vera Stoyanova, Gideon Dror, Jürgen Schmidhuber, Wayne Burleson, and Srinivas Devadas. Puf modeling attacks on simulated and silicon data. *IEEE Transactions on Information Forensics and Security (TIFS)*, 8(11), 2013.

[104] Georg T. Becker. The gap between promise and reality: On the insecurity of xor arbiter pufs. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2015.

[105] Georg T Becker. On the pitfalls of using arbiter-pufs as building blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 34 (8), 2015.

[106] Xiaolin Xu, Ulrich Rührmair, Daniel E. Holcomb, and Wayne Burleson. Security evaluation and enhancement of bistable ring pufs. In *Radio Frequency Identification*. Springer International Publishing, 2015.

[107] Qingli Guo, Jing Ye, Yue Gong, Yu Hu, and Xiaowei Li. Efficient attack on non-linear current mirror puf with genetic algorithm. In *IEEE Asian Test Symposium (ATS)*. IEEE, 2016.

[108] Arunkumar Vijayakumar, Vinay C Patil, Charles B Prado, and Sandip Kundu. Machine learning resistant strong puf: Possible or a pipe dream? In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016.

[109] Pranesh Santikellur, Aritra Bhattacharyay, and Rajat Subhra Chakraborty. Deep learning based model building attacks on arbiter puf compositions. Cryptology ePrint Archive, Report 2019/566, 2019. https://eprint.iacr.org/2019/566.

[110] Junye Shi, Yang Lu, and Jiliang Zhang. Approximation attacks on strong pufs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39 (10), 2019.

[111] Jeroen Delvaux and Ingrid Verbauwhede. Side channel modeling attacks on 65nm arbiter pufs exploiting cmos device noise. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2013.

[112] Durga Prasad Sahoo, Debdeep Mukhopadhyay, Rajat Subhra Chakraborty, and Phuong Ha Nguyen. A multiplexer-based arbiter puf composition with enhanced reliability and security. *IEEE Transactions on Computers (TC)*, 67(3), 2018.

[113] Phuong Ha Nguyen, Durga Prasad Sahoo, Chenglu Jin, Kaleel Mahmood, Ulrich RÃŒhrmair, and Marten van Dijk. The interpose puf: Secure puf design against state-of-the-art machine learning attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2019(4), 2019.

[114] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring puf: A new architecture for strong physical unclonable functions. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2011.

[115] Dieter Schuster and Robert Hesselbarth. Evaluation of bistable ring pufs using single layer neural networks. In *International Conference on Trust and Trustworthy Computing (TRUST)*. Springer International Publishing, 2014.

[116] Raghavan Kumar and Wayne Burleson. On design of a highly secure puf based on non-linear current mirrors. In *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2014.

[117] Arunkumar Vijayakumar and Sandip Kundu. A novel modeling attack resistant puf design based on non-linear voltage transfer characteristics. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2015.

[118] Fatemeh Ganji, Shahin Tajik, Fabian Fäßler, and Jean-Pierre Seifert. Strong machine learning attack against pufs with no mathematical model. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2016.

[119] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Controlled physical random functions. In *Annual Computer Security Applications Conference (ACSAC)*. IEEE, 2002.

[120] Anthony Van Herrewege, Stefan Katzenbeisser, Roel Maes, Roel Peeters, Ahmad-Reza Sadeghi, Ingrid Verbauwhede, and Christian Wachsmann. Reverse fuzzy extractors: Enabling lightweight mutual authentication for puf-enabled rfids. In *Financial Cryptography and Data Security (FC)*. Springer Berlin Heidelberg, 2012.

[121] Jing Ye, Yu Hu, and Xiaowei Li. Rpuf: Physical unclonable function with randomized challenge to resist modeling attack. In *IEEE Asian Hardware-Oriented Security and Trust (AsianHOST)*. IEEE, 2016.

[122] Mehrdad Majzoobi, Masoud Rostami, Farinaz Koushanfar, Dan S Wallach, and Srinivas Devadas. Slender puf protocol: A lightweight, robust, and secure authentication by substring matching. In *IEEE Symposium on Security and Privacy Workshops*. IEEE, 2012.

[123] Masoud Rostami, Mehrdad Majzoobi, Farinaz Koushanfar, Dan S Wallach, and Srinivas Devadas. Robust and reverse-engineering resilient puf authentication and key-exchange by substring matching. *IEEE Transactions on Emerging Topics in Computing (TETC)*, 2(1), 2014.

[124] Yansong Gao, Gefei Li, Hua Ma, Said F Al-Sarawi, Omid Kavehei, Derek Abbott, and Damith C Ranasinghe. Obfuscated challenge-response: A secure lightweight authentication mechanism for puf-based pervasive devices. In *IEEE International Conference on Pervasive Computing and Communication (PerCom) Workshops*. IEEE, 2016.

[125] Yansong Gao, Hua Ma, Said F Al-Sarawi, Derek Abbott, and Damith C Ranasinghe. Puf-fsm: A controlled strong puf. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(5), 2017.

[126] Klaus Kursawe, Ahmad-Reza Sadeghi, Dries Schellekens, Boris Skoric, and Pim Tuyls. Reconfigurable physical unclonable functions-enabling technology for tamper-resistant storage. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2009.

[127] Ulrich Rührmair, Christian Jaeger, and Michael Algasinger. An attack on puf-based session key exchange and a hardware-based countermeasure: Erasable pufs. In *Financial Cryptography and Data Security (FC)*. Springer Berlin Heidelberg, 2012.

[128] Ulrich Rührmair, Xiaolin Xu, Jan Sölter, Ahmed Mahmoud, Mehrdad Majzoobi, Farinaz Koushanfar, and Wayne Burleson. Efficient power and timing side channels for physical unclonable functions. In *Cryptographic Hardware and Embedded Systems (CHES)*. Springer Berlin Heidelberg, 2014.

[129] Georg T. Becker and Raghavan Kumar. Active and passive side-channel attacks on delay based puf designs. *IACR Cryptology ePrint Archive*, 2014. URL https://eprint.iacr.org/2014/287.

[130] Shahin Tajik, Heiko Lohrke, Fatemeh Ganji, Jean-Pierre Seifert, and Christian Boit. Laser fault attack on physically unclonable functions. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2015.

[131] Raghavan Kumar and Wayne Burleson. Hybrid modeling attacks on current-based pufs. In *IEEE International Conference on Computer Design (ICCD)*. IEEE, 2014.

[132] Jeroen Delvaux and Ingrid Verbauwhede. Fault injection modeling attacks on 65 nm arbiter and ro sum pufs via environmental changes. *IEEE Transactions on Circuits and Systems I: Regular Papers (TCSI)*, 61(6), 2014.

[133] Hongsik Park, Ali Afzali, Shu-Jen Han, George S Tulevski, Aaron D Franklin, Jerry Tersoff, James B Hannon, and Wilfried Haensch. High-density integration of carbon nanotubes via chemical self-assembly. *Nature Nanotechnology*, 7(12), 2012.

[134] SA Wolf, DD Awschalom, RA Buhrman, JM Daughton, von S von Molnár, ML Roukes, A Yu Chtchelkanova, and DM Treger. Spintronics: a spin-based electronics vision for the future. *Science*, 294(5546), 2001.

[135] R Stanley Williams. How we found the missing memristor. *IEEE Spectrum*, 45(12), 2008.

[136] Leon Chua. Memristor-the missing circuit element. *IEEE Transactions on Circuit Theory*, 18(5), 1971.

[137] Garrett S Rose, Jeyavijayan Rajendran, Harika Manem, Ramesh Karri, and Robinson E Pino. Leveraging memristive systems in the construction of digital logic circuits. *Proceedings of the IEEE*, 100(6), 2011.

[138] Shimeng Yu. Resistive random access memory (rram). *Synthesis Lectures on Emerging Engineering Technologies*, 2(5), 2016.

[139] Sung Hyun Jo, Ting Chang, Idongesit Ebong, Bhavitavya B Bhadviya, Pinaki Mazumder, and Wei Lu. Nanoscale memristor device as synapse in neuromorphic systems. *Nano letters*, 10(4), 2010.

[140] Yandan Wang, Wei Wen, Hai Li, and Miao Hu. A novel true random number generator design leveraging emerging memristor technology. In *Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2015.

[141] Mesbah Uddin, Md Sakib Hasan, and Garrett S. Rose. On the theoretical analysis of memristor based true random number generator. In *Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2019.

[142] Rui Liu, Huaqiang Wu, Yachun Pang, He Qian, and Shimeng Yu. A highly reliable and tamper-resistant rram puf: Design and experimental validation. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016.

[143] Mesbah Uddin, Md Badruddoja Majumder, Garrett S Rose, Karsten Beckmann, Harika Manem, Zahiruddin Alamgir, and Nathaniel C Cady. Techniques for improved reliability in memristive crossbar puf circuits. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2016.

[144] Yachuan Pang, Huaqiang Wu, Bin Gao, Dong Wu, An Chen, and He Qian. A novel puf against machine learning attack: Implementation on a 16 mb rram chip. In *IEEE International Electron Devices Meeting (IEDM)*. IEEE, 2017.

[145] Patrick Koeberl, Ünal Kocabaş, and Ahmad-Reza Sadeghi. Memristor pufs: a new generation of memory-based physically unclonable functions. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013.

[146] Garrett S Rose, Nathan McDonald, Lok-Kwong Yan, Bryant Wysocki, and Karen Xu. Foundations of memristor based puf architectures. In *IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 2013.

[147] Urbi Chatterjee, Rajat Subhra Chakraborty, Jimson Mathew, and Dhiraj K Pradhan. Memristor based arbiter puf: cryptanalysis threat and its mitigation. In *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*. IEEE, 2016.

[148] Rekha Govindaraj and Swaroop Ghosh. A strong arbiter puf using resistive ram within 1t-1r memory architecture. In *International Conference on Computer Design (ICCD)*. IEEE, 2016.

[149] Yansong Gao, Damith C Ranasinghe, Said F Al-Sarawi, Omid Kavehei, and Derek Abbott. mrPUF: A novel memristive device based physical unclonable function. In *International Conference on Applied Cryptography and Network Security (ACNS)*. Springer, 2015.

[150] Urbi Chatterjee, Rajat Subhra Chakraborty, Jimson Mathew, and Dhiraj K Pradhan. Memristor based arbiter puf: Cryptanalysis threat and its mitigation. In *International Conference on VLSI Design and International Conference on Embedded Systems (VLSID)*. IEEE, 2016.

[151] Rekha Govindaraj and Swaroop Ghosh. A strong arbiter PUF using resistive RAM within 1T-1R memory architecture. In *International Conference on Computer Design (ICCD)*. IEEE, 2016.

[152] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2008.

[153] Solar Designer. lpr LIBC RETURN exploit. http://insecure.org/sploits/linux.libc.return.lpr.sploit.html, 1997.

[154] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM conference on Computer and Communications Security*, 2007.

[155] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*. USENIX Association, 1998.

[156] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting systems from stack smashing attacks with stackguard. In *Linux Expo*. Citeseer, 1999.

[157] Hiroaki Etoh and Kunikazu Yoda. Propolice: Protecting from stack-smashing attacks. *Technical Report, IBM Research Division, Tokyo Research Laboratory*, 2000.

[158] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer Berlin Heidelberg, 2013.

[159] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. Dynamic canary randomization for improved software security. In *Annual Cyber and Information Security Research Conference (CISRC)*. ACM, 2016.

[160] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3), 2005.

[161] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2), 2008.

[162] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2009.

[163] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *ACM SIGSAC Conference on Computer & Communications Security (CCS)*. ACM, 2013.

[164] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2014.

[165] Qualcomm. Pointer authentication on ARMv8.3, 2017. URL https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[166] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *USENIX Security Symposium*. USENIX Association, 2019.

[167] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[168] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2004.

[169] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2013.

[170] Jeff Seibert, Hamed Okhravi, and Eric Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[171] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[172] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. Smashing the stack protector for fun and profit. In *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2018.

[173] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[174] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *USENIX Security Symposium*. USENIX Association, 2014.

[175] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[176] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A verified Hardware/Software Co-Design for remote attestation. In *USENIX Security Symposium*. USENIX Association, 2019.

[177] Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi. Cfinsight: A comprehensive metric for cfi policies. In *Network and Distributed Systems Security (NDSS) Symposium*, 2022.

[178] Tigist Abera, Raad Bahmani, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Matthias Schunter. Diat: Data integrity attestation for resilient collaboration of autonomous systems. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[179] Dongdong Huo, Yu Wang, Chao Liu, Mingxuan Li, Yazhe Wang, and Zhen Xu. Lape: A lightweight attestation of program execution scheme for bare-metal systems. In *IEEE International Conference on High Performance Computing and Communications; IEEE International Conference on Smart City; IEEE International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2020.

[180] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Tiny-cfa: Minimalistic control-flow attestation using verified proofs of execution. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021.

[181] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. ScaRR: Scalable runtime remote attestation for complex systems. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. USENIX Association, 2019.

[182] Yumei Zhang, Xinzhi Liu, Cong Sun, Dongrui Zeng, Gang Tan, Xiao Kan, and Siqi Ma. Recfa: Resilient control-flow attestation. In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2021.

[183] Dwaine Clarke, Srinivas Devadas, Marten van Dijk, Blaise Gassend, and G Edward Suh. Incremental multiset hash functions and their application to memory integrity checking. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Springer Berlin Heidelberg, 2003.

[184] Ghada Dessouky, Tigist Abera, Ahmad Ibrahim, and Ahmad-Reza Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018.

[185] Munir Geden and Kasper Rasmussen. Hardware-assisted remote runtime attestation for critical embedded systems. In *International Conference on Privacy, Security and Trust (PST)*. IEEE, 2019.

[186] Saar Drimer. Volatile FPGA design security–a survey. *IEEE Computer Society Annual Volume*, 2008.

[187] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Physical Unclonable Functions and public-key crypto for FPGA IP protection. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2007.

[188] Saar Drimer. Security for volatile FPGAs. Technical report, University of Cambridge, Computer Laboratory, 2009.

[189] Hirak Kashyap and Ricardo Chaves. Compact and on-the-fly secure dynamic reconfiguration for volatile FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(2), 2016.

[190] Nisha Jacob, Jakob Wittmann, Johann Heyszl, Robert Hesselbarth, Florian Wilde, Michael Pehl, Georg Sigl, and Kai Fischer. Securing FPGA SoC configurations independent of their manufacturers. In *IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2017.

[191] Florian Unterstein, Nisha Jacob, Neil Hanley, Chongyan Gu, and Johann Heyszl. Sca secure and updatable crypto engines for FPGA SoC bitstream decryption. In *ACM Workshop on Attacks and Solutions in Hardware Security (ASHES) Workshop*. ACM, 2019.

[192] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. Fpga-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7, 2018.

[193] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding performance differences of fpgas and gpus. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.

[194] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2014.

[195] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking: SmartNICs in the public cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018.

[196] Microsoft Research. Project Brainwave. https://www.microsoft.com/en-us/research/project/project-brainwave.

[197] Yan Xu, Olivier Muller, Pierre-Henri Horrein, and Frédéric Pétrot. Hcm: an abstraction layer for seamless programming of DPR FPGA. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012.

[198] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. FPGAs in the cloud: booting virtualized hardware accelerators with openstack. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2014.

[199] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the cloud. In *ACM Conference on Computing Frontiers (CF)*. ACM, 2014.

[200] Suhaib A Fahmy, Kizheppatt Vipin, and Shanker Shreejith. Virtualized FPGA accelerators for efficient cloud computing. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015.

[201] Jagath Weerasinghe, Francois Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in hyperscale data centers. In *IEEE Intl Conf on Ubiquitous Intelligence and Computing and IEEE Intl Conf on Autonomic and Trusted Computing and IEEE Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 2015.

[202] Oliver Knodel, Patrick Lehmann, and Rainer G Spallek. RC3E: reconfigurable accelerators in data centres and their provision by adapted service models. In *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2016.

[203] Huawei Cloud. FPGA accelerated cloud server. https://www.huaweicloud.com/en-us/product/facs.html.

[204] Furkan Turan and Ingrid Verbauwhede. Trust in fpga-accelerated cloud computing. *ACM Computing Surveys (CSUR)*, 53(6), 2020.

[205] Jonas Krautter, Dennis RE Gnad, Falk Schellenberg, Amir Moradi, and Mehdi B Tahoori. Active fences against voltage-based side channels in multi-tenant FPGAs. In *International Conference On Computer Aided Design (ICCAD)*. IEEE/ACM, 2019.

[206] Jonas Krautter, Dennis RE Gnad, and Mehdi B Tahoori. Mitigating electrical-level attacks towards secure multi-tenant FPGAs in the cloud. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 12(3), 2019.

[207] Tuan Minh La, Kaspar Matas, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. FPGADefender: malicious self-oscillator scanning for Xilinx UltraScale+ FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 13(3), 2020.

[208] Ken Eguro and Ramarathnam Venkatesan. FPGAs for trusted cloud computing. In *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012.

[209] Muhammad ES Elrabaa, Mohammed Al-Asli, and Marwan Abu-Amara. Secure computing enclaves using FPGAs. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 18(2), 2019.

[210] *Intel Stratix 10 Device Security User Guide*. Intel Corporation, 12 2021.

[211] *User Guide SmartFusion2 and IGLOO2 FPGA Security and Best Practices*. Microsemi, 2019. Rev. 6.

[212] Dan Boneh and Matthew Franklin. An efficient public key traitor tracing scheme. In *Annual International Cryptology Conference (CYPTO)*. Springer Berlin Heidelberg, 1999.

[213] Thien Duc Nguyen, Phillip Rieger, Hossein Yalame, Helen Möllering, Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Ahmad-Reza Sadeghi, Thomas Schneider, et al. Flguard: Secure and private federated learning. *arXiv preprint arXiv:2101.02281*, 2021.

ABOUT THE AUTHOR

Shaza Zeitouni is a research assistant at the Technische Universität Darmstadt, Germany. In 2006, she received her B.Sc. in Electronic Engineering from Aleppo University, Syria. In 2014, she received her M.Sc. in Computational Science from the Technische Universität Braunschweig, Germany. Her research has been focused on hardware-assisted security architectures and trust anchors: the security of Physically Unclonable Functions, the design of hardware-based trust anchors for runtime protection on embedded systems and trusted configuration and computing on Field Programmable Gate Arrays.

AWARDS

- CROSSING Collaboration Award 2019
- CROSSING Collaboration Award 2016

ACADEMIC ACTIVITIES

Sub and external reviewer in top tier hardware and security conferences and journals:

- Design Automation Conference (DAC) 2015, 2016, 2018, 2019 and 2021.
- International Conference On Computer Aided Design (ICCAD), 2016, 2017, 2018, 2019.
- International Conference on Applied Cryptography and Network Security (ACNS) 2016 and 2018.
- IEEE European Symposium on Security & Privacy (Euro S&P) 2020.
- IEEE Symposium on Security & Privacy (S&P) 2018.
- ACM Conference on Computer and Communications Security (CCS) 2015, 2016 and 2019.
- ACM ASIA Conference on Computer and Communications Security (AsiaCCS) 2015, 2018 and 2019.
- Design Automation and Test in Europe Conference (DATE) 2021.
- ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec) 2016.
- IEEE International Symposium on Hardware Oriented Security and Trust (HOST) 2015 and 2016.

- Financial Cryptography and Data Security (FC) 2016.
- International Symposium on Research in Attacks, Intrusions and Defenses (RAID) 2019.
- International Conference on Hardware/Software Co Design and System Synthesis (CODES + ISSS) 2020.
- ACM Transactions on Design Automation of Electronic Systems (TODAES).
- ACM Transactions on Privacy and Security (TOPS).
- IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD).

PEER-REVIEWED PUBLICATIONS

Hossein Fereidooni, Samuel Marchal, Markus Miettinen, Azalia Mirhoseini, Helen Möllering, Thien Duc Nguyen, Phillip Rieger, Ahmad-Reza Sadeghi, Thomas Schneider, Mohammad Hossein Yalame, Shaza Zeitouni. SAFELearn: Secure Aggregation for private FEderated Learning. In *Proceeding of the 4th Deep Learning and Security Workshop* (DLS'21), 2021.
Ghada Dessouky, Ahmad-Reza Sadeghi, Shaza Zeitouni. SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities. In *Proceedings of the 6th IEEE European Symposium on Security and Privacy* (EuroS&P'21), 2021.

Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, Nele Mentens. Trusted Configuration in Cloud FPGAs. In *Proceedings of the 29th IEEE International Symposium On Field-Programmable Custom Computing Machines* (FCCM'21), 2021.

Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, Ahmad-Reza Sadeghi. On the Security of Strong Memristor-based Physically Unclonable Functions. In *Proceedings of the 57th ACM/IEEE Design Automation Conference* (DAC'20), 2020.

Johannes Buchmann, Ghada Dessouky, Tommaso Frassetto, Ágnes Kiss, Ahmad-Reza Sadeghi, Thomas Schneider, Giulia Traverso, Shaza Zeitouni. SAFE: A Secure and Efficient Long-Term Distributed Storage System. In *Proceedings of the 8th International Workshop on Security in Blockchain and Cloud Computing* (SBC'20), 2020.

Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, Ahmad-Reza Sadeghi. CHASE: Configurable Hardware-Assisted Security Extension for Real-Time Systems. In: *Proceedings of the 38th ACM/IEEE International Conference On Computer Aided Design* (ICCAD'19), 2019.

Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, Ahmad-Reza Sadeghi. HardScope: Hardening Embedded Systems Against Data-Oriented Attacks. In *Proceedings of the 56th ACM/IEEE Design Automation Conference* (DAC'19), 2019.

Shaza Zeitouni, David Gens, Ahmad-Reza Sadeghi. It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs. In *Proceedings of the 55th ACM/IEEE Design Automation Conference* (DAC'18), 2018.

Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, Ahmad-Reza Sadeghi. ATRIUM: Runtime Attestation Resilient Under Memory Attacks. In *Proceedings of the 36th ACM/IEEE International Conference On Computer Aided Design* (ICCAD'17), 2017.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Shaza Zeitouni. SeED: Secure Non-Interactive Attestation for Embedded Devices. In *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (WiSec'17), 2017.

Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, Ahmad-Reza Sadeghi. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *Proceedings of the 54th ACM/IEEE Design Automation Conference* (DAC'17), 2017.

Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, Michael Zohner. Pushing the communication barrier in secure computation using lookup tables. In *the 24th Annual Network and Distributed System Security Symposium* (NDSS'17), 2017.

Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, Shaza Zeitouni. DARPA: Device Attestation Resilient to Physical Attacks. In *Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (WiSec'16), 2016.

Ebrahim Songhori, Shaza Zeitouni, Ghada Dessouky, Thomas Schneider, Ahmad-Reza Sadeghi, Farinaz Koushanfar. GarbledCPU: A MIPS Processor for Secure Computation in Hardware. In *Proceedings of the 53th ACM/IEEE Design Automation Conference* (DAC'16), 2016.

Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, Ahmad-Reza Sadeghi. Remanence Decay Side-Channel: The PUF Case. In *IEEE Transactions on Information Forensics and Security* (TIFS), Vol. 11, 2015.

Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni. Automated Synthesis of Optimized Circuits for Secure Computation. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security* (CCS'15), 2015.

Erklärung gemäß §9 der Promotionsordnung

Hiermit versichere ich, die vorliegende Dissertation selbstständig und nur unter Verwednung der angegebenen Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 2022

Shaza Zeitouni

A

REMANENCE DECAY SIDE-CHANNEL: THE PUF CASE

[1] Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, Ahmad-Reza Sadeghi. "Remanence Decay Side-Channel: The PUF Case (TIFS'15)". In IEEE Transactions on Information Forensics and Security (TIFS), Vol. 11, 2015.

Remanence Decay Side-Channel: The PUF Case

Shaza Zeitouni, Yossef Oren, Christian Wachsmann, Patrick Koeberl, and Ahmad-Reza Sadeghi

***Abstract*— We present a side-channel attack based on remanence decay in volatile memory and show how it can be exploited effectively to launch a noninvasive cloning attack against SRAM physically unclonable functions (PUFs)—an important class of PUFs typically proposed as lightweight security primitives, which use existing memory on the underlying device. We validate our approach using SRAM PUFs instantiated on two 65-nm CMOS devices. We discuss countermeasures against our attack and propose the constructive use of remanence decay to improve the cloning resistance of SRAM PUFs. Moreover, as a further contribution of independent interest, we show how to use our evaluation results to significantly improve the performance of the recently proposed TARDIS scheme, which is based on remanence decay in SRAM memory and used as a time-keeping mechanism for low-power clockless devices.**

***Index Terms*— SRAM PUF, fault injection attack, side-channel analysis, data remanence decay.**

I. Introduction

PHYSICALLY Unclonable Functions (PUFs) have been an attractive research area and are increasingly proposed as building blocks in cryptographic protocols and security architectures. One major class of PUFs and the focus of this paper are memory-based PUFs [1]–[6]. These PUFs are commonly proposed as an alternative to secure non-volatile storage and are used in a variety of anti-counterfeiting mechanisms and authentication schemes [1], [8]–[12]. Today, PUF-based security products are already on the market, mainly targeting IP-protection and anti-counterfeiting applications as well as Radio-Frequency Identification (RFID) systems [14] and [15].

Memory-based PUFs are arrays of volatile memory elements, such as SRAM cells [1], [5], flip-flops [2], [6], or latches [3], [4]. These elements are typically bi-stable circuits with two stable states corresponding to a logical zero and one. By controlling the voltage level at the inputs of the elements, they enter either one of the two states. Due to the bi-stability, the elements retain their states as long as they are supplied with power. Memory-based PUFs exploit the following phenomenon: When powering up such an element without applying any voltage at the bit-line input, its state mainly depends on the physical characteristics of the underlying transistors. Due to uncontrollable manufacturing variations, these characteristics are unique for each physical instantiation of the element. Hence, the state of all memory elements, after powering the memory without applying any voltage at the bit-lines, can be used as a unique identifier, known as *'PUF response'*, of the device containing the memory. However, since the response of a memory-based PUF could be read out and copied to another device, protecting the PUF response against unauthorized accesses is considered a fundamental requirement of memory-based PUF implementations, which implies mimimally the presence of some security mechanism to prevent unauthorized accesses to the PUF response.

Memory-based PUFs are considered to be cost-effective since it is possible to use the already existing memory of the device as PUFs [1], [7], [10], [14]–[17]. However, in this case the memory is also used to store data of some other components in the device and will be overwritten at some point of time. In particular, volatile memory is typically initialized before it is used for data storage. Further, although volatile memory loses the data it stores when it is powered off, the data are not immediately lost but rather decay slowly over time [18], [19]. Hence, it is very likely that any data written to the memory of a memory-based PUF may affect the PUF response when removing or reducing the power supply for short time periods. Although this effect has been discussed in the literature [5], [23]–[25] it has never been used to attack memory-based PUFs. A preliminary and shorter version of this work has been published at [22]. This is an extended version that includes new evaluation results of voltage-based attacks.

A. Contribution

We present, to the best of our knowledge, the first side-channel attack based on the remanence decay in volatile memory and show how it can be exploited for a non-invasive cloning attack against SRAM-based PUFs. In particular, our contributions are as follows:

1) First Cloning Attack on SRAM PUFs Using Remanence Decay Side Channels: Our attack recovers the secret response of a memory-based PUF in applications where the underlying memory is overwritten with a known value after the PUF response has been read. This attack can be applied to all memory-based PUF systems that share the PUF memory with some other functionality, which is often suggested in the

Manuscript received March 26, 2015; revised September 17, 2015 and November 24, 2015; accepted December 6, 2015. Date of publication December 25, 2015; date of current version March 16, 2016. This work was supported in part by the European Commission through the UNIQUE Project under Grant ICT-2007-238811 and in part by the Deutsche Forschungsgemeinschaft through the CROSSING Project under Grant CRC 1119. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Stefan Mangard.

S. Zeitouni and P. Koeberl are with the Intel Collaborative Research Institute for Secure Computing, Darmstadt 64293, Germany (e-mail: shaza.zeitouni@trust.cased.de; patrick.koeberl@intel.com).

Y. Oren is with Ben-Gurion University of the Negev, Beersheba 8410501, Israel (e-mail: yos@bgu.ac.il).

C. Wachsmann is with Intel Deutschland GmbH, Nürnberg 90449, Germany (e-mail: christian.wachsmann@intel.com).

A.-R. Sadeghi is with Technische Universität Darmstadt, Darmstadt 64289, Germany (e-mail: ahmad.sadeghi@trust.tu-darmstadt.de).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIFS.2015.2512534

literature to allow cost-effective PUF implementations [1], [7], [10], [14]–[17]. We show that the attack is successful against small memory-based PUFs even when using common lab equipment.

2) Experimental Validation of the Attack: We validate the feasibility of our attack using SRAM PUFs instantiated on two 65nm CMOS devices, and suggest several improvements to increase the performance of our attack.

3) Constructive Use of Remanence Decay: We propose using remanence decay as a source of side-channel information to enhance the cloning-resistance of SRAM PUFs. Cloning such a PUF would require emulating the remanence decay behavior, which increases the cost of cloning or even render it uneconomical. We also propose a time-memory tradeoff to dramatically reduce the complexity of the recently proposed TARDIS [19] time-keeping mechanism for clockless devices from linear to logarithmic time, enhancing its applicability to many practical scenarios; we further propose a simplified version of TARDIS which requires only constant time and minimal non-volatile storage.

B. Outline

We introduce our notation, the system and the adversary model in Section II. The attack is described in Section III and its experimental validation is presented in Section IV. A practical instantiation of our attack is shown in Section V. We discuss the impact and improvements of the attack in Section VI and make suggestions on the constructive use of remanence decay, including the improved TARDIS algorithm, in Section VII. We give an overview of the related work in Section VIII and finally conclude in Section IX.

II. Model and Preliminaries

We consider devices that contain memory-based PUFs, where the underlying memory can be overwritten with a known value after the PUF response has been read. This typically happens when the PUF memory is also used for data storage by some other functionality in the device, which is a common approach to cost-effective implementations of memory-based PUFs [1], [7], [10], [14]–[17].

A. Initial State

Volatile memory is typically initialized, i.e., overwritten with a specific bit pattern (usually all zeroes or ones), before it is used as a data storage. We denote this pattern as the *initial state* of the memory.

Definition 1 (Initial State): The initial state *of the memory is the matrix $\vec{M}_{\text{init}}$ representing the data that is written to the memory before it is used as a data storage.*

B. Start-Up State

Data stored in the volatile memory are not immediately lost when the power to the memory is removed or decreased but decay slowly over time [18], [19]. Hence, when powered off only for a short time, the memory may still hold some of the data that have been written to it before the power cycle.

We capture this aspect by introducing the notion of the *start-up state*.

Definition 2 (Start-Up State): Let υ_{nom} be the nominal supply voltage of the memory V_{dd}.

Consider the following experiment:

1) *Set supply voltage of memory elements to 0 V for time t*
2) *Set supply voltage of memory elements to υ_{nom}*
3) *Read the states of all memory elements and store them in a matrix $\vec{M}_t$*

We say that $\vec{M}_t$ is the start-up state *of the memory with respect to time t.*

Further, consider the following experiment:

1) *Set supply voltage of memory elements to $\upsilon < \upsilon_{\text{nom}}$ for constant time τ*
2) *Set supply voltage of memory elements to υ_{nom}*
3) *Read the states of all memory elements and store them in a matrix $\vec{M}_\upsilon$*

We say that $\vec{M}_\upsilon$ is the start-up state *of the memory with respect to voltage υ and a constant time τ.*

C. PUF State

The response of a memory-based PUF corresponds to the start-up state of the underlying memory, where the memory has been powered off long enough that any data previously stored in it have decayed. We capture this aspect by introducing the notion of the *PUF state* of a memory.

Definition 3 (PUF State): Let t_∞ indicates the time required for previously stored data to completely decay from the memory. We denote the start-up state $\vec{M}_{t_\infty}$ as the PUF state *$\vec{M}_{\text{PUF}}$ of the memory, i.e., $\vec{M}_{\text{PUF}} := \vec{M}_{t_\infty}$.*

In the case where the memory has been powered off only for a short time before it is used as a PUF, the PUF response may be distorted by the data previously stored in the memory.

D. Device Behavior

At some point while the device is running, it reads the start-up state of its memory and uses it as the PUF response in some computation. In many applications the result of this computation can be observed from outside the device. For instance, in PUF-based (authentication) protocols [7], [10], [11], the device receives some query Q and responds with a message X that depends on the PUF response. In these schemes, the response of the memory-based PUF is typically used to derive a cryptographic secret that is used to compute X. However, the device behavior is not limited to challenge-response protocols. In the extreme case X could be only one single bit of information, e.g., indicating whether the correct PUF response was extracted from the memory or not. For instance, in PUF-based IP protection schemes [1], [8], [9], the device refuses to boot in the case where the PUF response is incorrect, which can be observed by the adversary. We capture this aspect by introducing the notion of *device behavior*.

Definition 4 (Device Behavior): Let $\vec{M}$ be the start-up state of the device memory with respect to some time t or voltage υ (Definition 2). Further, let Q be some query that can be sent to the device. We denote with $X = \mathsf{Dev}(\vec{M}, Q)$ the response to Q of the device using the start-up state $\vec{M}$. The *algorithm* Dev *describes the* behavior *of the device with respect to Q and $\vec{M}$.*

E. Assumptions and Adversary Model

Following the common adversary model of memory PUFs [1], [7], [10], [14]–[17], we assume that the adversary $\mathcal{A}$ cannot simply read the plain PUF response from the underlying memory. This means that $\mathcal{A}$ does *not* know the start-up state $\vec{M}$ (Definition 2) with respect to any time t or voltage v and, in particular, $\mathcal{A}$ does not know the PUF state $\vec{M}_{\text{PUF}}$ (Definition 3). Further, we assume that all algorithms implemented in the device are known to $\mathcal{A}$ (Kerckhoffs' principle). This means that $\mathcal{A}$ could compute $X = \mathsf{Dev}(\vec{M}, Q)$ if he knew $\vec{M}$ and Q. Moreover, $\mathcal{A}$ knows the initial state $\vec{M}_{\text{init}}$ (Definition 1) that is part of the algorithms used by the device. Furthermore, we assume that $\mathcal{A}$ can observe the device behavior (Definition 4) and that $\mathcal{A}$ can control the time t the memory is powered off before it is used as a PUF as well as the supply voltage v of the memory. This means that $\mathcal{A}$ can send some query Q to the device and observe its reaction X that depends on the device's start-up state $\vec{M}$. We also assume that the fuzzy extractor generates an output for any arbitrary *state* $\vec{M}$ that differs from $\vec{M}_{\text{PUF}}$ with more than t bits, which the fuzzy extractor can correct.

III. Cloning Attack Using Remanence Decay

The high level idea and approach of our attack is to recover the PUF response in a device that overwrites the SRAM of the PUF with some data that are known to the adversary $\mathcal{A}$ (cf. Section II). The attack principle is similar to Biham-Shamir attack [23] to extract a secret key stored in some device (e.g., a smart card).

The Biham-Shamir attack consists of two phases. In the first phase, $\mathcal{A}$ collects a sequence of ciphertexts, each encrypting the same plaintext with a slightly different key. More detailed, $\mathcal{A}$ requests the device to encrypt the plaintext and, after receiving the corresponding ciphertext, he injects a fault into the device that sets one bit of the key to a known value. $\mathcal{A}$ repeats this step until all bits in the key are set to known values. In the second phase of the attack, $\mathcal{A}$ iteratively recovers the secret key of the device, starting from the last ciphertext that has been generated by the device using the key known to $\mathcal{A}$. $\mathcal{A}$ performs an exhaustive search for each key used by the device to generate each ciphertext collected in the first phase. Since the keys of two consecutive ciphertexts differ in at most one single bit and the value of this bit is known to $\mathcal{A}$, this exhaustive search is linear in the bit-length of the key. This way, $\mathcal{A}$ can recover the secret key of the device with a total effort quadratic in the bit-length of the key.

Similar to the Biham-Shamir attack, we iteratively collect a series of device responses to the same query, each generated using a different start-up state. In each iteration, we send the query to the device, record its response (that depends on the start-up state), and then inject a fault to change some bits in the start-up state. The fault injection is performed by carefully controlling the amount of remanence decay undergone by the SRAM, e.g., by increasing the time the device is powered off between two iterations, or by reducing the device's supply voltage for a fixed time period. Due to the different remanence decay exhibited by each SRAM cell, for any given power-off period or reduction of voltage supply, some SRAM cells will lose the known value of the initial state and revert back to their unknown PUF state, some will retain their initial state and some will exhibit *metastable* behaviour by taking a random state. Hence, in contrast to the Biham-Shamir attack, the number of bits k that are different in the start-up states used in two consecutive iterations is typically larger than one bit. However, as we show in Section IV, k has an upper bound that highly depends on the exhaustive method or the computational power and a lower bound imposed by the accuracy of the equipment used to control the remanence decay.

In the second phase of the attack, we iteratively recover the PUF state. A trivial approach would be to perform a simple exhaustive search for all cells that have reverted to their PUF state in the start-up states of two consecutive iterations of phase one. However, while this approach works for small values of k, it is inefficient for large values of k. In Section VI we discuss several approaches to reduce the value of k by improving the test setup and to reduce the complexity of the search for the changed bit positions. Before we describe our attack in detail, we explain the underlying requirements and building blocks.

A. Controlling the Remanence Decay

An essential requirement for our attack is that the adversary $\mathcal{A}$ can precisely control the remanence decay in the SRAM. There are two approaches how this can be achieved. The *voltage-based* approach directly decreases the supply voltage to the chip for a certain amount of time τ, while the *time-based* approach sets the supply voltage of the chip to 0 V for a precisely-measured amount of time t. In general, the time-based approach is easier to use since it only requires a precise timer to trigger the voltage drop, while the voltage-based approach requires an expensive precision DC power source. Next, we present results for both approaches.

B. Data Remanence Experiment

One major building block of our attack is the data remanence experiment where the adversary $\mathcal{A}$ observes how the remanence decay affects the behavior of the device containing the PUF.

Definition 5 (Data Remanence Experiment): Consider a device that overwrites the memory used by the PUF with some known data. Let v_{nom} be the nominal supply voltage of the device. Let $\vec{M}_{\text{PUF}}$ (Definition 3) be the PUF state and $\vec{M}_{\text{init}}$ (Definition 1) be the initial state of the device memory. Further, let Dev *(Definition 4) be the algorithm describing the device behavior with respect to some start-up state $\vec{M}_t$ or $\vec{M}_v$ (Definition 2).*

The time-based data remanence experiment $X = \overline{\mathsf{DRE}}(\vec{M}_{\text{init}}, t, Q)$ *is as follows:*

1) *Set the memory content of the device to $\vec{M}_{\text{init}}$*
2) *Temporarily set the supply voltage of the device to 0 V for time t and then set it back to v_{nom}*

67

3) *Send the query Q to the device and observe its response* $X = \mathsf{Dev}(\vec{M}_t, Q)$

Further, we define the voltage-based data remanence experiment $X = \mathsf{DRE}(\vec{M}_{\text{init}}, \upsilon, Q)$ *as follows:*

1) *Set the memory content of the device to* $\vec{M}_{\text{init}}$
2) *Temporarily set the supply voltage of the device to* $\upsilon < \upsilon_{\text{nom}}$ *for a constant time* τ *and then set it back to* υ_{nom}
3) *Send the query Q to the device and observe its response* $X = \mathsf{Dev}(\vec{M}_\upsilon, Q)$

C. Finder Algorithm

Another building block of our attack is the finder algorithm, which recovers the PUF state based on the device behavior observed in a series of data remanence experiments.

Definition 6 (Finder Algorithm): Let $\vec{M}_{i+1}$ and $\vec{M}_i$ be two start-up states (Definition 2) that consist of n bits and that differ in at most $k < n$ bits, i.e., the Hamming distance $\mathsf{dist}(\vec{M}_i, \vec{M}_{i+1}) \leq k$. *Further, let* $X_{i+1} = \mathsf{Dev}(\vec{M}_{i+1}, Q)$ *for some arbitrary device query Q. A* finder algorithm *is a probabilistic polynomial time algorithm* $\mathsf{Finder}(\vec{M}_i, Q, X_{i+1})$ *that returns* $\vec{M}_{i+1}$.

The finder is most efficient when $\mathsf{dist}(\vec{M}_i, \vec{M}_{i+1})$ is minimal, ideally one. In this case, Finder can recover an unknown n-bit start-up state $\vec{M}_{i+1}$ from $\vec{M}_i$ and X_{i+1} by performing a simple exhaustive search with linear complexity in n. However, $\mathsf{dist}(\vec{M}_i, \vec{M}_{i+1})$ is typically larger than one since multiple SRAM cells may have similar remanence decay behavior, decay at the same time/voltage, while other SRAM cells may be metastable (i.e., take a random value) [19], [24]–[26]. In the worst case, where up to k bits have changed in a start-up state with n bits, a trivial finder performing an exhaustive search may require up to $\sum_{\ell=1}^{k} \binom{n}{\ell}$. Typically n is a fixed system parameter while k can be somehow controlled or reduced as we discuss in Section VI.

D. Details of the Attack

The attack is detailed in Algorithm 1 on the example of the time-based approach and works as follows. The adversary $\mathcal{A}$ chooses an arbitrary device query Q (Step 1) and records the response X_{PUF} generated by the device using the PUF state $\vec{M}_{\text{PUF}}$ (Step 2). Then, $\mathcal{A}$ performs a series of time-based DRE experiments (Definition 5) where he slightly increases the power-off time t_i used in each experiment (Steps 3 and 4).[1] This way, $\mathcal{A}$ obtains a sequence of device responses $X_1, \ldots, X_f$ to the same query Q generated by the device using the start-up states $\vec{M}_{t_1}, \ldots, \vec{M}_{t_f}$, respectively, where $\mathsf{dist}(\vec{M}_{t_i}, \vec{M}_{t_{i+1}})$ for all $1 \leq i \leq (f-1)$ is upper bounded by some value k. Observe that $\vec{M}_{t_0} = \vec{M}_{\text{init}}$ is the initial state (Definition 1) and $\vec{M}_{t_f} = \vec{M}_{\text{PUF}}$ is the PUF state (Definition 3) of the SRAM. Next, $\mathcal{A}$ uses the Finder algorithm (Definition 6) to iteratively recover $\vec{M}_{\text{PUF}}$ from the device responses observed in Steps 3 to 4. Specifically,

[1]An adversary using the voltage-based approach would gradually lower the supply voltage (for a fixed amount of time) instead of increasing the power-off time.

Algorithm 1 Extracting the PUF State of an SRAM PUF-Enabled Device (Time-Based Approach)

Consider a device that uses the same SRAM for the PUF and some other functionality. Let $\vec{M}_{\text{init}}$ be the initial state (Definition 1) and t_∞ be the decay time (cf. Definition 3) of the device memory. Further, let Δt be the difference between the power-off times used in two consecutive time-based DRE experiments (cf. Definition 5). Further, let i and f be indices. The attack works as follows:

```
1) Fix an arbitrary device query $Q$
2) Record $X_{\text{PUF}} = \mathsf{DRE}(\vec{M}_{\text{init}}, t_\infty, Q)$
3) Set $i \leftarrow 0$ and $t_0 = 0$
4) Repeat:
    a) Set $i \leftarrow i + 1$
    b) Set $t_i = t_{i-1} + \Delta t$
    c) Record $X_i = \mathsf{DRE}(\vec{M}_{\text{init}}, t_i, Q)$
    d) Stop when $X_i = X_{\text{PUF}}$ and set $f = i$
5) Set $i \leftarrow 0$ and $\vec{M}_{t_0} = \vec{M}_{\text{init}}$
6) Repeat:
    a) Set $i \leftarrow i + 1$
    b) Compute $\vec{M}_{t_i} = \mathsf{Finder}(\vec{M}_{t_{i-1}}, Q, X_i)$
    c) Stop when $i = f$
7) Return $M_{t_f}$
```

starting from the known initial state $\vec{M}_{t_0} = \vec{M}_{\text{init}}$, the adversary iteratively recovers each $\vec{M}_{t_{i+1}}$ from $\vec{M}_{t_i}$ and X_{i+1} until $\mathcal{A}$ arrives at the PUF state $\vec{M}_{t_f} = \vec{M}_{\text{PUF}}$.

Theorem 1 (Success of the Attack): The attack in Algorithm 1 successfully recovers the PUF state $\vec{M}_{\text{PUF}}$. The worst case complexity of the attack when using a trivial Finder *algorithm (Definition 6) is* $f \cdot \sum_{\ell=1}^{k} \binom{n}{\ell}$, *where f is the number of* DRE *experiments (cf. Definition 5), n is the size of the SRAM, and k is the maximum Hamming distance of the start-up states $\vec{M}_{t_i}$ and $\vec{M}_{t_{i+1}}$ used by the device in two consecutive* DRE *experiments for all* $1 \leq i \leq (f-1)$.

The complexity of the attack strongly depends on the value of k, which highly depends on the accuracy of the equipment and method used to control the remanence decay in the SRAM. Typical values are $k = 0.1469 \cdot n$ for the time-based approach and $k = 0.1004 \cdot n$ for the voltage-based approach (cf. Section IV). Moreover, in our experiments for the time-based approach we observed a decay time of $t_\infty = 2,000\ \mu s$ and used $\Delta t = 1\ \mu s$, resulting in $f = \lceil 2,000\ \mu s / 1\ \mu s \rceil = 2,000$. For the *voltage-based* approach we observed remanence decay for supply voltages between $0.4\ V$ and $0\ V$ and used $\Delta\upsilon = 2\ mV$, resulting in $f = \lceil 400\ mV / 2\ mV \rceil = 200$. Proof of Theorem 1 can be found in [22].

IV. Experimental Validation of the Attack

In order to reduce the complexity of the attack, it is required that only a small number k of SRAM cells in two consecutive DRE experiments change their states during the transition from the known (initial) state to the final PUF state. This number k is mainly controlled by two factors: (1) the accuracy

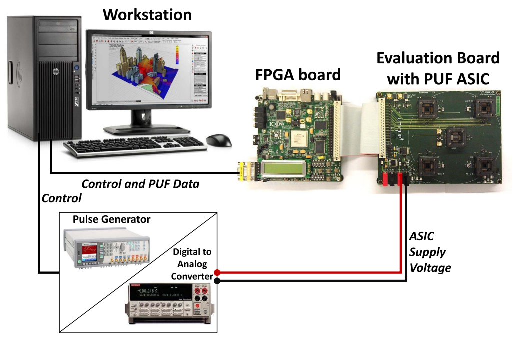


Fig. 1. Test setup for the time-based/voltage-based approaches using an Agilent pulse generator/Keithley sourcemeter.

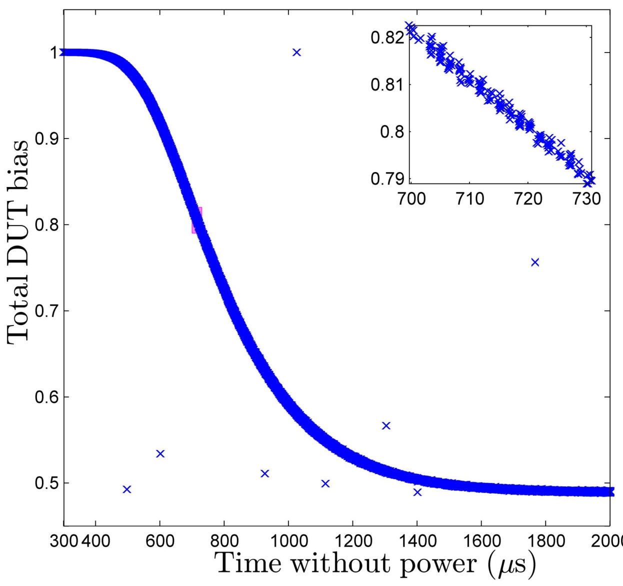


Fig. 2. Chip-scale view of remanence decay (time-based approach).

of the equipment used to control the remanence decay of the memory during the attack and (2) the number of metastable SRAM cells, i.e., those that take random states. In this section, we investigate the impact of both factors on the remanence decay in the SRAM PUFs implemented in two 65 nm CMOS ASICs. Our evaluation uses both the time-based and the voltage-based approach to control the remanence decay.

A. Test Setup

Our analysis is based on data obtained from two ASICs that have been manufactured in TSMC 65 nm CMOS technology within an Europractice multi-project wafer run. The ASIC has been designed within the UNIQUE[2] research project. Each ASIC implements four different SRAM PUF instances, each using 8 kBytes of SRAM. The test setup consists of an ASIC evaluation board, a Xilinx Virtex 5 FPGA board, a power supply; either an Agilent 81150 pulse/function/arbitrary pulse generator for the time-based approach or a Keithley 2400 general-purpose sourcemeter for the voltage-based approach, and a workstation (Figure 1). The evaluation board allows controlling the ASIC supply voltage using an external power supply. In each experiment, we wrote a pre-determined bit pattern (i.e., all ones) to the SRAM, used the pulse generator or sourcemeter to deliver a temporary voltage drop with precisely controlled width and amplitude and finally read back the memory contents of the SRAM. The rated accuracy of the Agilent 81150 pulse generator has a temporal resolution of 5 ns and an amplitude resolution of 25 mV. The Keithley 2400 sourcemeter has a basic accuracy of 50 μV.

To accelerate and simplify the remanence decay process, we did not place any decoupling capacitors between the pulse generator/sourcemeter's output and the ASIC's supply voltage input, as shown in [19], the effect of such a capacitor in the time-based approach is an increase in the time between the power-down event and the beginning of remanence decay. In the voltage-based approach, the capacitor increases the time required for the ASIC's power input to converge to the required value, again resulting in a slowdown of the remanence decay process. The interaction with the evaluation board and the ASICs is performed by the FPGA, which is connected to a workstation that controls the PUF evaluation and the pulse generator or the sourcemeter. Further, the workstation is used to process and store the data obtained from the ASICs. The tests with the Keithley sourcemeter were performed in a refrigerator at temperatures between 2.7 °C and 7.6 °C, while tests with the Agilent 81150 pulse generator were performed at room temperature (approx. 25°C) in an air conditioned laboratory because we wanted to capture the effect of power-off time on data remanence without controlling the ambient temperature, as it is already known that some attacks use low temperatures to decelerate data remanence of SRAM cells [18].

[2]http://www.unique-project.eu/

B. Chip-Scale Modeling

The purpose of this experiment was to observe and to reproduce the decay behaviour reported in [19] and to gauge its stability and reproducibility for the SRAM PUF for the time-based and the voltage-based approach.

1) Time-Based Approach: A series of 10,000 data remanence experiments with an initial state $\vec{M}_{\text{init}}$ consisting of only ones was performed. Each experiment was repeated 10 times with 1,000 different power-off times t between 300 μs and $2,000$ μs. During the power-off time the supply voltage was set to 0 V. After each experiment we measured for each SRAM cell the probability that it still stores the value we wrote to it before the power cycle. We call this probability the *bias* of the cell.

2) Voltage-Based Approach: We performed 30 series of data remanence experiments with an initial state $\vec{M}_{\text{init}}$ consisting of only ones. Each series consists of 201 experiments, where the voltage was dropped by 2 mV for 1 ms starting at 0.4 V, as the experiments show no decay to zero in the range between 1.2 V and 0.4 V, and then set back to the default supply voltage of 1.2 V. For each experiment, we measured the probability that each SRAM cell preserves the value written to it before the power cycle.

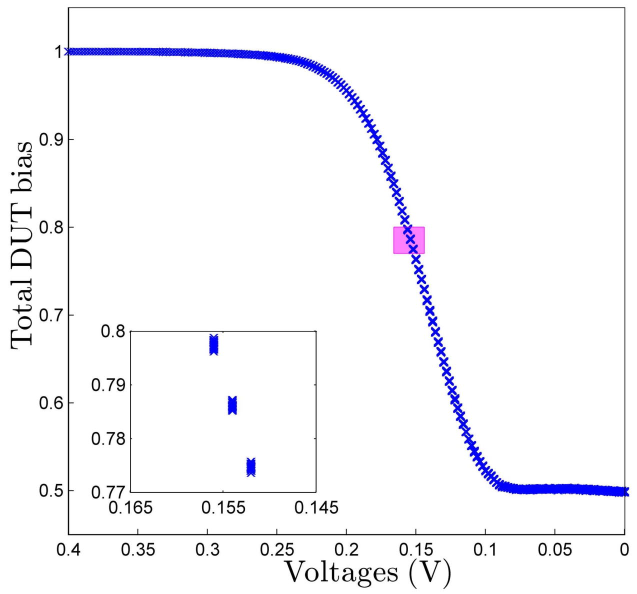


Fig. 3. Chip-scale view of remanence decay (voltage-based approach).

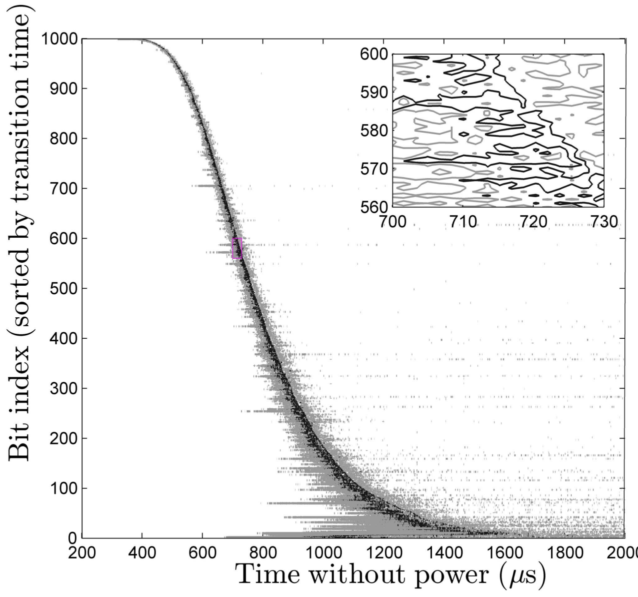


Fig. 4. Bit-scale view of remanence decay (time-based approach).

3) Chip-Scale Results: Our results are depicted in Figure 2 and Figure 3. In both figures, the y-axis corresponds to the mean bias over all SRAM cells, while the x-axis corresponds to the total *time* the ASIC was without power in Figure 2 or to the *voltages* applied to the ASIC Figure 3. Each cross in the graphs corresponds to a single experiment. As shown in Figure 2, the average bias over all SRAM cells decays very reliably from 1 to the expected 0.5 [25], [26] during the course of 2 ms, while in Figure 3 exhibits the average bias over all SRAM cells decaying from 1 to 0.5 in the range of 0.4 V to 0 V. The results are also compatible with the findings in [21] and [33], i.e., the typical values of DRV fall in the range between 80 mV and 250 mV.

As the zoomed-in views in Figure 2 and Figure 3 show that, there is a small variation in the measured bias between identical experiments, which was either due to the physical limitations of our test setup or due to those SRAM cells exhibiting metastability.

C. Bit-Scale Modeling

This experiment investigates whether the individual SRAM cells have different transition times (voltages), which is required in our attack. With the *transition time* (resp. *transition voltage*) of an SRAM cell we mean the point in time (resp. voltage level) where the cell loses the value that has been written to it and reverts to its PUF state. Based on the results of the previous experiment, we estimated the bias of each SRAM cell over time.

1) Bit-Scale Results: Figure 4 and Figure 5 display 2-D contour plots of the cell-level behaviour of the SRAM PUF. Again, the inner graphs represent zoomed-in portions of the graphs. Each horizontal row in the graph corresponds to the bias of a single SRAM cell, selected out of 1000 representative cells whose final bias were close to zero. We only selected cells with final bias close to zero since the cells with a final

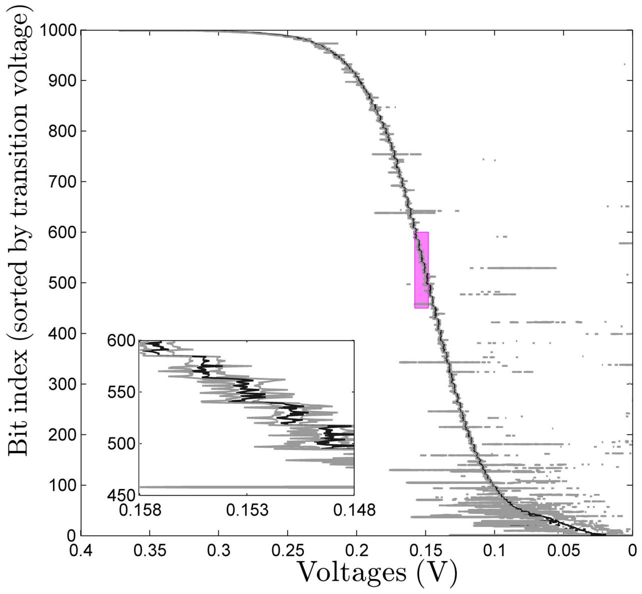


Fig. 5. Bit-scale view of remanence decay (voltage-based approach).

bias close to one will not show any decay behavior in our experiment where we wrote a logical one to all memory cells before the power cycle. For the purpose of legibility, the cells were sorted in the graphs by their transition times (voltages). The left and right gray lines on the graphs correspond to times in Figure 4 and voltages in Figure 5, when the bias of each bit is one and zero, respectively. The black line corresponds to the time (voltage) when the bias of each bit is 0.5.

As shown in Figure 4, each individual SRAM cell has a different remanence decay time surrounded by a short period of metastability in which the cell may enter both states. The median metastability period measured was 30 μs and the worst-case metastability rate was 14%. Figure 5

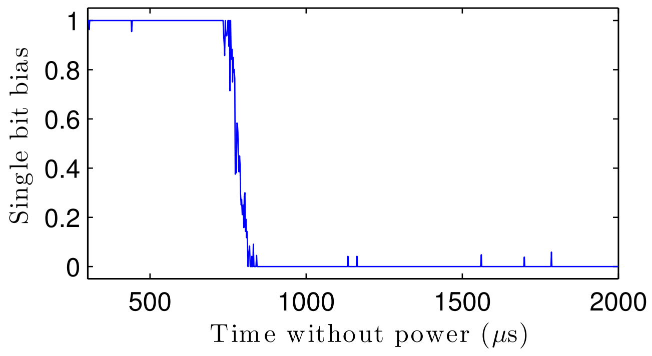


Fig. 6. Close-up look at a single bit for the time-based approach.

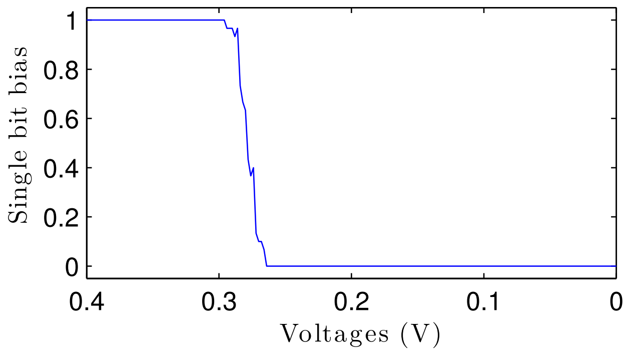


Fig. 7. Close-up look at a single bit for the voltage-based approach.

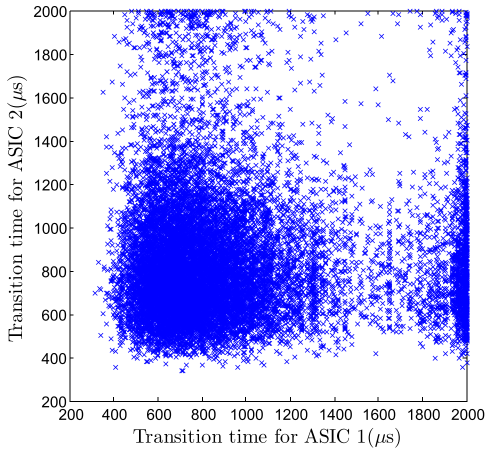


Fig. 8. Correlation between the transition time in two different devices.

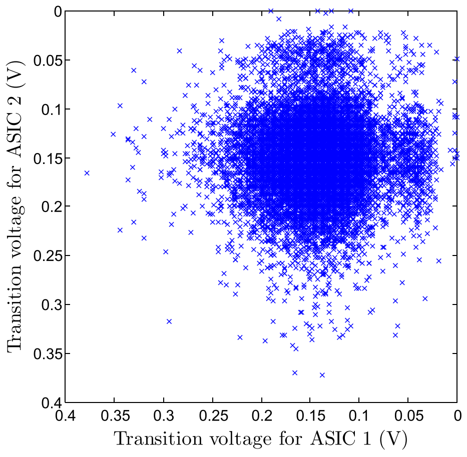


Fig. 9. Correlation between the transition voltage in two different devices.

shows each individual SRAM cell has a different remanence decay voltage with a voltage band on either side exhibiting metastability. The median metastability voltage was 4 mV and the worst-case metastability rate was 10%, which is 28% lower than the time-based result. In general, the maximum size of a PUF that can be attacked using our methodology is limited by the metastability, as we discuss further in Section VI.

A detailed look at the evolution of the bias of a single bit over time and voltage is shown in Figure 6 and Figure 7 respectively, and the small spikes which can be noted in the otherwise monotonic plots are probably the result of noise encountered when measuring the bit in its metastable state.

D. Cross-Device Comparison

Next we investigated whether the transition times (resp. voltages) of the SRAM cells in one device allow to infer some information on the transition times (resp. voltages) of the SRAM cells in another device. A second goal of this experiment was to get a first impression of whether the transition times (voltages) in SRAM cells could be used to identify individual SRAM chips, an idea we discuss in Section VI. In this experiment, we measured the bias over time and the transition times (voltages) of each SRAM cell in two ASICs. Again, we considered only cells whose PUF state is zero.

1) Cross-Device Results: The results for the time-based approach as well as voltage-based approach are shown in Figure 8 and Figure 9 respectively. Each cross in the graphs corresponds to the bias of a single SRAM cell. In both figures, the x-coordinate of each point is the transition time (voltage) of the SRAM cell on the first ASIC, while the y-coordinate is the transition time (voltage) of the same SRAM cell on the second ASIC. The behavior in Figure 8 is due the fact that the number of metastable cells in time-based experiment is larger than those in voltage-based experiment. Besides that, in time-based experiment, some of the cells, which decay to zero, oscillate over power cycles before they settle down on zero in the last power cycle. Figure 8 also shows that, the transition times of the two ASICs are virtually uncorrelated, which we confirmed by computing the normalized cross correlation ρ of both data sets, which is $\rho = 0.002$. Our results are in line with the findings by Holcomb et al. [21] who also suggest using the remanence decay behaviour as a source of unique information to identify individual devices. Figure 9 also confirms the same conclusion, i.e. the transition voltages of the two ASICs are uncorrelated and the normalized cross correlation ρ of both data sets $\rho = 0.0012$.

E. Time-Based vs. Voltage-Based Attacks

The evaluation results in the previous sections confirm results in the literature [19], [27] and show that the voltage-based approach is less sensitive to temperature variations,

TABLE I
COMPARISON OF VOLTAGE-BASED AND TIME-BASED REMANENCE

Remanence control	Voltage-based	Time-based
Bits stable at 1	44.82%	43.45%
Bits stable at 0	45.14%	41.86%
Metastability rate (worst case)	10.04%	14.69%

making it potentially more effective in an attack than the time-based approach. Our results are summarized in Table I, which shows that using the voltage-based approach results in a significantly lower metastability rate than using the time-based approach. This means that a voltage-based attack may still be effective in situations where the time-based attack fails. An interesting observation is that the set of metastable SRAM cells in voltage-based approach shows 28% improvement over time-based approach, which indicates that most of the inaccuracies in our experiments are due to the limitations of our test setup and not due the physical properties of the SRAM PUF itself.

V. Effectiveness of the Attack in Practical Settings

To investigate the effectiveness of our attack in a practical setting, we created a standard implementation of an SRAM PUF-based authentication scheme. This scheme uses a standard secret-key-based challenge-response protocol and derives the underlying key from the PUF response using a basic repetition code [28].[3] In more detail, during the enrollment of the device, the memory addresses of those 128 SRAM bytes whose PUF state is highly biased (i.e., that have a Hamming weight of 0, 1, 7, or 8) are stored as the public helper data, each representing one bit of the secret key stored in the PUF. The key is reconstructed from the PUF as follows. The 128 SRAM bytes whose addresses are stored in the helper data are read from the SRAM and the value of each bit in the key is set as the result of a simple majority voting over all bits in the respective byte to ensure that the 128 bits key derived from the 128 highly-biased SRAM bytes are stable. Even though the bits participating in the majority voting (and their associated transition times and voltages) are spatially averaged, there is still a single point of time/voltage for each output bit where a majority of its constituent bits transit to the PUF state. The bit-flips before this transition period are absorbed by the error-correcting code and can be ignored by the attacker. This is a simple example of the general behaviour of error correcting codes, where the output symbol changes only after a sufficient number of errors has accumulated. The resulting secret key K is then used in the secret key-based challenge-response protocol, i.e. $X = \mathsf{MAC}_K(Q)$, where MAC is a Message Authentication Code. The attack is as in Section III-D. However, we use an optimized Finder algorithm (Definition 6) that only searches for key candidates with a Hamming distance less than 10 bits from the previous key, which significantly improves the performance of the attack compared to the trivial Finder described in Definition 6. The overall running time of the attack is estimated as $2^{53.6}$ MAC operations. Considering that modern CPUs can perform 2^{31} AES operations per second [29], [30], the total cost of the attack on an AES-based MAC is $2^{22.6}$ CPU-seconds, or approximately two CPU-months. The attack can easily be parallelized by testing multiple key candidates simultaneously, making the attack even more practical for moderately-funded adversaries.

[3]We omit the linear encoding used in [28] and the privacy amplification typically used in PUF-based key storage since it has no effect on our attack.

VI. Impact of Our Attack and Countermeasures

A. Impact

Our results in Section IV show that by carefully controlling the power-off times or the supply voltage of the SRAM PUF, one can reliably control the number of metastable bits as required by the attack described in Section III. This means that, even if we use the trivial finder algorithm discussed in Definition 6, common lab equipment and the less effective time-based approach to control the remanence decay in the SRAM, we can recover a 216-bit SRAM PUF derived key by making at most 2^{64} calls to the Dev algorithm (cf. Definition 4). Using the voltage-based approach with the same finder algorithm and equipment as in the time-based approach, we can extract the response of a 315-bit SRAM PUF derived key in the same time. Further, our results in Section V show that, depending on the post-processing of the PUF responses, our attack can also be applied to systems using larger PUFs. Hence, it is problematic to overwrite the memory of an SRAM PUF with a known value, which, however, is required when the PUF memory is also used for other purposes, as suggested in many prior works [1], [7], [10], [14]–[17]. This particularly holds for resource-constrained devices with only small amounts of SRAM, such as RFIDs or medical implants [7], [10], [14], where SRAM PUFs without shared memory are impractical.

B. Improving the Attack

One approach to lower the complexity of our attack is using more accurate equipment that allows a very precise control of the remanence decay in the SRAM using the voltage-based approach, which limits the number of metastable bits and the complexity of the finder algorithm (cf. Definition 6).

Furthermore, several optimizations of the finder algorithm are possible: The order in which the individual SRAM cells transition from their initial state to their PUF state is different for the time-based and the voltage-based approach (cf. Section V). Further, in some scenarios the adversary may be able to control the initial state of the SRAM. This results in four different ways to observe the decay behavior of each SRAM cell and allows the adversary to choose the way with the lowest metastability rate for his attack, which can significantly reduce the complexity of the naïve finder algorithm (cf. Definition 6).

Another approach to improve the complexity of the finder algorithm is to take advantage of the algorithms used by the device to process the PUF responses (cf. Section V). These algorithms typically include an error correction mechanism such as a fuzzy extractor [31], which helps to maintain the consistency of the PUF response to the same challenge under different environmental variations affecting the underlying physical object. Due to this error correction the device response changes only when the error correction mechanism fails to correct *state* $\vec{M}$ which differs from $\vec{M}_{\mathrm{PUF}}$ with more than t bits, which the error correction mechanism can handle. Hence, the finder algorithm needs to consider only one single candidate of each codeword class. This can either be done explicitly by considering the structure of the error correcting code or by casting the problem as an optimization problem and using an optimizer [32].

C. Countermeasures

There are several countermeasures that prevent our attack by breaking the underlying assumptions but that are impractical in low-resource scenarios such as RFIDs and sensors [7], [10], [14]. One approach to prevent the attack described in Section III is using an additional memory that can only be accessed by the PUF. However, this contradicts the idea of using the existing memory of the device and significantly increases implementation costs. Another approach is to wait until any value stored in the memory has decayed before reading the PUF response. However, this requires the device to have some notion of time and increases the boot time, which can be problematic in some applications. Further, the attack can be prevented by designing the algorithms processing the PUF response such that the device behavior for different start-up states is indistinguishable by the adversary. However, this might imply the use of complex cryptographic primitives for authentication schemes that exceed the capabilities of *resource-constrained* devices for which PUFs with shared memory have been proposed [7], [10], [14].

VII. Constructive Use of Remanence Decay

A. Device Authentication

The remanence decay behavior can be used to authenticate an SRAM to some verifier. Specifically, using the same approach as in our attack, a verifier could force the SRAM into a partially reverted state by, e.g., writing some value to the SRAM and then powering the device off for a carefully controlled amount of time. Since the verifier knows the (secret) PUF state of the SRAM and the decay behavior of the genuine device, he can determine the partially reverted SRAM state and check whether it matches the expected state of the SRAM to be authenticated. Care must be taken that this additional functionality does not expose the device to our attack, for example by requiring that the verifier successfully authenticates to the device before being allowed to write to the SRAM.

Note that, it is much more difficult to clone such an SRAM PUF since the clone must emulate the SRAM decay behavior, which requires the clone to contain a time-keeping mechanism that raises its costs. Our results suggest that for an SRAM of size n bits there are $\log(n!)$ bits of entropy encoded in the *order* at which individual SRAM cells revert to their PUF state. However, further evaluations are needed to assess the practicality of this approach, in particular the temperature-dependency and the effect of aging on the decay behavior must be investigated.

B. Improving the TARDIS Time-Keeping Algorithm

The use of SRAM remanence decay has been proposed as a time-keeping mechanism for clockless low-power devices, such as passive RFID tags [19]. This mechanism, called TARDIS, allows a clockless device to estimate how much time has passed since its last power-down and aims to impede oracle attacks. TARDIS consists of two main elements: the Init algorithm which sets all SRAM cells to a fixed value (all ones) and the Decay algorithm which determines how long the device has been without power based on the number of ones that are still stored in the SRAM. These two operations consume a non-negligible amount of power and add 15.2 ms to the start-up time of the device.

Our observations on the behaviour of remanence decay can be used to dramatically improve the performance of the TARDIS system. As our results show, the transition time of each bit is uniquely determined by its individual DRV. By profiling the SRAM in an offline phase, we can thus determine the *order* in which the SRAM cells return to their PUF state and store this ordering in the non-volatile memory of the device. Now, if we observe that a certain group of bits has reverted to its PUF state, we can infer that all bits which have a lower transition time have also returned to their PUF state. Similarly, if a certain group of bits is still in its initial state then all bits that have a longer transition time are also still in their initial state. Knowing this ordering, we can replace the linear-time Decay algorithm of [19] with the well known binary search algorithm that takes logarithmic time. To deal with metastability, the algorithm should sample not only one but a group of bits for each transition time period.

If the device needs to detect only whether or not the *entire* SRAM has returned to its PUF state, another improvement is possible that dramatically decreases the running time of both the Init and the Decay algorithms from linear time to constant time. In this case, both algorithms need only to access those SRAM cells that are known to be the *last* to revert to the PUF state. Since most of the applications described in [19] can be adapted to use this improvement, our results enhance the applicability of the TARDIS system to practical scenarios. We stress that the SRAM used by the TARDIS scheme cannot be used as an SRAM PUF since its content is well-known in this case.

VIII. Related Work

While the impact of remanence decay on the randomness that can be extracted from SRAM cells and the reliability of SRAM PUFs has been discussed in the literature [5], [23]–[25] it has never been used as a side channel to attack SRAM PUFs. In fact, some papers investigate side channel attacks in the context of PUFs, mainly focusing on the side channel leakage of the algorithms processing the PUF response [33], [34] or proposing combinations of side channels attacks on PUFs with modeling or fault injection attacks [35]–[37]. The impact of environmental changes on the repeatability of PUF response has been introduced as a source of fault injection attack on arbiter and RO PUFs in [38], and current-based PUFs in [39], while the same impact has been evaluated in [30] and [31] for memory-based PUFs, however no results on fault injection attacks have been reported. In contrast, to the best of our knowledge, we present the first cloning attack that injects faults into the SRAM PUF and uses the data remanence effect in SRAM as a side channel to recover the (secret) PUF response.

It has been shown that SRAM PUFs can be emulated and physically cloned. By physically inspecting the SRAM hardware the adversary learns information that helps emulating the PUF [40]. Further, it has been shown that after learning the response of an SRAM PUF p_1, a focussed ion beam (FIB) can be used to modify the circuits of the SRAM cells of another SRAM PUF p_2 so that p_2 shows a very similar challenge/response behavior as p_1 [41].

Data remanence in DRAM has been used to extract security-sensitive data from the random access memory of PCs and workstations [18]. While these attacks aim to recover some data that has been written to an unprotected memory, the goal of our attack is to recover the start-up pattern of an SRAM PUF that is typically protected by some kind of access control mechanism.

IX. Conclusion

We demonstrated a simple non-invasive cloning attack on SRAM PUFs using remanence decay as a side-channel and validated its feasibility on 8 KBytes SRAM PUFs instantiated on two 65 nm CMOS devices. Our attack and evaluation is general and can be optimized for concrete systems. Our evaluation results show that even without optimizations, attacks on small SRAM PUFs are feasible using common lab equipment. We discussed countermeasures against our attack and suggest using remanence decay to improve the cloning-resistance of SRAM PUFs. We showed how our evaluation results can be used to improve the performance of TARDIS [19], a timekeeping mechanism for clockless devices.

We investigated both the time-based approach and the voltage-based approach to control the data remanence decay in the SRAM. Our results show that the voltage-based approach is more promising than the time-based approach. Directions for future work include the design of non-trivial finder algorithms that, e.g., exploit the properties of the algorithms used by the device processing the PUF response.

Acknowledgements

The author thank Ünal Kocabaş for preparing the lab experiments in the first phase of this work.

References

[1] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Physical unclonable functions and public-key crypto for FPGA IP protection," in *Proc. Int. Field Program. Logic Appl. (FPL)*, 2007, pp. 189–195.

[2] R. Maes, P. Tuyls, and I. Verbauwhede, "Intrinsic PUFs from flip-flops on reconfigurable devices," in *Proc. 3rd Benelux Workshop Inf. Syst. Secur.*, 2008, pp. 1–17.

[3] Y. Su, J. Holleman, and B. P. Otis, "A digital 1.6 pJ/bit chip identification circuit using process variations," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 69–77, Jan. 2008.

[4] S. S. Kumar, J. Guajardo, R. Maes, G.-J. Schrijen, and P. Tuyls, "Extended abstract: The butterfly PUF protecting IP on every FPGA," in *Proc. IEEE Int. Workshop Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2008, pp. 67–70.

[5] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Trans. Comput.*, vol. 58, no. 9, pp. 1198–1210, Sep. 2009.

[6] V. van der Leest, G.-J. Schrijen, H. Handschuh, and P. Tuyls, "Hardware intrinsic security from D flip-flops," in *Proc. 5th ACM Workshop Scalable Trusted Comput. (ACM STC)*, 2010, pp. 53–62.

[7] P. Tuyls and L. Batina, "RFID-tags for anti-counterfeiting," in *Topics in Cryptology* (Lecture Notes in Computer Science), vol. 3860. Springer, 2006, pp. 115–131.

[8] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Brand and IP protection with physical unclonable functions," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2008, pp. 3186–3189.

[9] J. A. Roy, F. Koushanfar, and I. L. Markov, "Ending piracy of integrated circuits," *Computer*, vol. 43, no. 10, pp. 30–38, 2010.

[10] A.-R. Sadeghi, I. Visconti, and C. Wachsmann, "Enhancing RFID security and privacy by physically unclonable functions," in *Towards Hardware-Intrinsic Security* (Information Security and Cryptography). Springer, 2010, pp. 281–305.

[11] I. Eichhorn, P. Koeberl, and V. van der Leest, "Logically reconfigurable PUFs: Memory-based secure key storage," in *Proc. 6th ACM Workshop Scalable Trusted Comput. (ACM STC)*, 2011, pp. 59–64.

[12] Intrinsic ID. (2013). *Product Webpage*. [Online]. Available: http://www.intrinsic-id.com/products.htm

[13] *NXP Strengthens SmartMX2 Security Chips With PUF Anti-Cloning Technology*, NXP Semiconductors, Eindhoven, The Netherlands, 2013.

[14] J. Guajardo, M. Asim, and M. Petković, "Towards reliable remote healthcare applications using combined fuzzy extraction," in *Towards Hardware-Intrinsic Security* (Information Security and Cryptography). Springer, 2010, pp. 387–407.

[15] S. Kardaş, M. S. Kiraz, M. A. Bingöl, and H. Demirci, "A novel RFID distance bounding protocol based on physically unclonable functions," in *RFID. Security and Privacy* (Lecture Notes in Computer Science). Springer, Jun. 2011.

[16] P. Koeberl, J. Li, A. Rajan, C. Vishik, and W. Wu, "A practical device authentication scheme using SRAM PUFs," in *Trust and Trustworthy Computing* (Lecture Notes in Computer Science), vol. 6740. Springer, Jun. 2011, pp. 63–77.

[17] P. Koeberl, J. Li, R. Maes, A. Rajan, C. Vishik, and M. Wójcik, "Evaluation of a PUF device authentication scheme on a discrete 0.13 μm SRAM," in *Proc. 3rd Int. Conf. Trusted Syst. (INTRUST)*, vol. 7222. 2012, pp. 271–288.

[18] J. A. Halderman *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, no. 5, pp. 91–98, May 2009.

[19] A. Rahmati, M. Salajegheh, D. Holcomb, J. Sorber, W. P. Burleson, and K. Fu, "TARDIS: Time and remanence decay in SRAM to implement secure protocols on embedded devices without clocks," in *Proc. USENIX Secur. Symp.* 2012, pp. 221–236.

[20] N. Saxena and J. Voris. (Jul. 2009). "We can remember it for you wholesale: Implications of data remanence on the use of RAM for true random number generation on RFID tags (RFIDSec 2009)." [Online]. Available: http://arxiv.org/abs/0907.1256

[21] D. E. Holcomb, A. Rahmati, M. Salajegheh, W. P. Burleson, and K. Fu, "DRV-fingerprinting: Using data retention voltage of SRAM cells for chip identification," in *Radio Frequency Identification. Security and Privacy Issues* (Lecture Notes in Computer Science), vol. 7739, J.-H. Hoepman and I. Verbauwhede, Eds. Springer, 2013, pp. 165–179.

[22] Y. Oren, A.-R. Sadeghi, and C. Wachsmann, "On the effectiveness of the remanence decay side-channel to clone memory-based PUFs," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), vol. 8086. Springer, 2013, pp. 107–125.

[23] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology* (Lecture Notes in Computer Science), vol. 1294. Springer, 1997, pp. 513–525.

[24] D. E. Holcomb, W. P. Burleson, and K. Fu, "Initial SRAM state as a fingerprint and source of true random numbers for RFID tags," in *Proc. Conf. Workshop RFID Secur. (RFIDSec)*, Jul. 2007, pp. 1–12.

[25] M. Bhargava, C. Cakir, and K. Mai, "Comparison of bi-stable and delay-based physical unclonable functions from measurements in 65 nm bulk CMOS," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, Sep. 2012, pp. 1–4.

[26] S. Katzenbeisser, Ü. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "PUFs: Myth, fact or busted? A security evaluation of physically unclonable functions (PUFs) cast in silicon," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), vol. 7428. Springer, 2012, pp. 283–301.

[27] H. Qin, Y. Cao, D. Markovic, A. Vladimirescu, and J. Rabaey, "SRAM leakage suppression by minimizing standby supply voltage," in *Proc. 5th Int. Symp. Quality Electronic Design (ISQED)*, 2004, pp. 55–60.

[28] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls, "Efficient helper data key extractor on FPGAs," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), vol. 5154. Berlin, Germany: Springer, Jul. 2008, pp. 181–197.

[29] Calomel.org. *AES-NI SSL Performance Study*. [Online]. Available: https://calomel.org/aesni_ssl_performance.html, accessed 2015.

[30] Intel. *AES-NI Performance Enhancements: Hytrust Datacontrol Case Study*. [Online]. Available: https://software.intel.com/en-us/articles/intel-aes-ni-performance-enhancements-hytrust-datacontrol-case-study, accessed 2015.

[31] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," in *Advances in Cryptology* (Lecture Notes in Computer Science), vol. 3027. Springer, May 2004, pp. 523–540.

[32] Y. Oren, M. Renauld, F.-X. Standaert, and A. Wool, "Algebraic side-channel attacks beyond the hamming weight leakage model," in *Cryptographic Hardware and Embedded Systems* (Lecture Notes in Computer Science), vol. 7428, E. Prouff and P. Schaumont, Eds. Springer, 2012, pp. 140–154.

[33] D. Karakoyunlu and B. Sunar, "Differential template attacks on PUF enabled cryptographic devices," in *Proc. IEEE Int. Workshop Inf. Forensics Secur. (WIFS)*, Dec. 2010, pp. 1–6.

[34] D. Merli, D. Schuster, F. Stumpf, and G. Sigl, "Side-channel analysis of PUFs and fuzzy extractors," in *Trust and Trustworthy Computing* (Lecture Notes in Computer Science), vol. 6740. Springer, Jun. 2011, pp. 33–47.

[35] A. Mahmoud, U. Rührmair, M. Majzoobi, and F. Koushanfar. *Combined Modeling and Side Channel Attacks on Strong PUFs*. [Online]. Available: http://eprint.iacr.org/2013/632, accessed 2013.

[36] U. Rührmair *et al.*, "Efficient power and timing side channels for physical unclonable functions," in *Cryptographic Hardware and Embedded Systems*, vol. 8731. Berlin, Germany: Springer, 2014, pp. 476–492.

[37] G. T. Becker and R. Kumar, *Active and Passive Side-Channel Attacks on Delay Based PUF Designs*. [Online]. Available: http://eprint.iacr.org/2014/287, accessed 2014.

[38] J. Delvaux and I. Verbauwhede, "Fault injection modeling attacks on 65 nm arbiter and RO sum PUFs via environmental changes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 61, no. 6, pp. 1701–1713, Jun. 2014.

[39] R. Kumar and W. Burleson, "Hybrid modeling attacks on current-based PUFs," in *Proc. 32nd IEEE Int. Conf. Comput. Design (ICCD)*, Oct. 2014, pp. 493–496.

[40] D. Nedospasov, J.-P. Seifert, C. Helfmeier, and C. Boit, "Invasive PUF analysis," in *Proc. Fault Diagnosis Tolerance Cryptogr. (FDTC)*, Aug. 2013, pp. 30–38.

[41] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert, "Cloning physically unclonable functions," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*, Jun. 2013, pp. 1–6.

Shaza Zeitouni received the bachelor's degree in electronics engineering from the University of Aleppo, and the master's degree in computational sciences in engineering from TU Braunschweig. She is currently pursuing the Ph.D. degree with the Intel Collaborative Research Institute for Secure Computing, TU Darmstadt. She a Research Assistant. Her research interests include hardware security, physically unclonable functions, and secure privacy preserving computation.

Yossef Oren received the M.Sc. degree in computer science from the Weizmann Institute of Science, and the Ph.D. degree in electrical engineering from Tel-Aviv University. He is a Senior Lecturer with the Department of Information Systems Engineering, Ben Gurion University (BGU), and a member of the BGU's Cyber Security Research Center. He was a Postdoctoral Research Scientist with the Network Security Laboratory, Columbia University, NY, and a member of the Security Laboratory with Samsung Research Israel. His research interests include secure hardware (power analysis and other hardware attacks and countermeasures, and low-resource cryptographic constructions for lightweight computer) and cryptography in the real world (consumer and voter privacy in the digital era, and Web application security).

Christian Wachsmann received the Ph.D. degree in computer science from Technische Universität Darmstadt, Germany. He is a Security Architect with Intel Deutschland GmbH. Before joining Intel and at the time of writing, he was a Postdoctoral Researcher with the Intel Collaborative Research Institute for Secure Computing, TU Darmstadt. He is the main author of more than 30 scientific publications in internationally renowned journals and conferences in information and communications security. His research focuses on the design, development, formal modeling, and security analysis of security architectures and cryptographic protocols to verify the software integrity (attestation) of embedded systems.

Patrick Koeberl received the bachelor's degree in electronics engineering from the University of Ulster, and the master's degree in electronic systems from Dublin City University. He is a Senior Security Architect with Intel Labs focusing on mobile and embedded security. He is with the Intel Collaborative Research Institute for Secure Computing, Darmstadt, Germany, where he leads research into hardware primitives for resource constrained environments. Before joining Intel, he has held positions, including Technical Lead with Mentec Computer Systems, working on processor emulation hardware for deployment in siemens carrier grade telephony switches, the Head of Hardware Development with AEP Networks, where he led the development of enterprise hardware security modules certified to FIPS140-2 Level 4 and architected the AEP10K public-key coprocessor ASIC, subsequently acquired by Intel and marketed as the IXCP210. More recently, he was the CTO of Dajeil and led the effort to develop an XML offload engine for deployment in Web services security. He holds 15 granted or pending patents and has authored more than 15 peer-reviewed scientific publications in international conferences and journals.

Ahmad-Reza Sadeghi received the Ph.D. degree in computer science with the focus on privacy protecting cryptographic protocols and systems from the University of Saarland, Saarbrücken, Germany. He worked in Research and Development of Telecommunications enterprises, amongst others Ericson Telecommunications. He is a Full Professor of Computer Science with Technische Universität Darmstadt (TU Darmstadt). He is the Head of the System Security Laboratory with the Center for Advance Security Research Darmstadt, and the Director of the Intel Collaborative Research Institute for Secure Computing with TU Darmstadt. He served on the Editorial Board of the *ACM Transactions on Information and System Security*, and as a Guest Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN (Special Issue on Hardware Security and Trust). Currently, he is the Editor-in-Chief of the *IEEE Security and Privacy Magazine*, and on the Board of ACM Books. He has received the renowned German prize Karl Heinz Beckurts for his research on trusted and trustworthy computing technology and its transfer to industrial practice. The award honors excellent scientific achievements with high impact on industrial innovations in Germany.

B

IT'S HAMMER TIME: HOW TO ATTACK (ROWHAMMER-BASED) DRAM-PUFS

[2] Shaza Zeitouni, David Gens, Ahmad-Reza Sadeghi. "It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs". In Proceedings of the 55th ACM/IEEE Design Automation Conference (DAC'18), 2018.

It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs

Shaza Zeitouni
Technische Universität Darmstadt
Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

David Gens
Technische Universität Darmstadt
Darmstadt, Germany
david.gens@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Physically Unclonable Functions (PUFs) are still considered promising technology as building blocks in cryptographic protocols. While most PUFs require dedicated circuitry, recent research leverages DRAM hardware for PUFs due to its intrinsic properties and wide deployment. Recently, a new memory-based PUF was proposed that utilizes the infamous Rowhammer effect in DRAM. In this paper, we show two *remote* attacks on DRAM-based PUFs. First, a DoS attack that exploits the Rowhammer effect to manipulate PUF responses. Second, a modeling attack that predicts PUF responses by observing few challenge-response pairs. Our results indicate that DRAM may not be suitable for PUFs.

KEYWORDS

PUFs, Rowhammer, Remote attacks

ACM Reference Format:
Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. 2018. It's Hammer Time: How to Attack (Rowhammer-based) DRAM-PUFs. In *DAC '18: DAC '18: The 55th Annual Design Automation Conference 2018, June 24–29, 2018, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3195970.3196065

1 INTRODUCTION

Physically Unclonable Functions (PUFs) have been considered a promising technology to establish trust anchors in embedded systems with minimal hardware requirements. PUFs allow for utilizing inherent manufacturing process variations to extract unique but reproducible secrets/identifiers, which are generated through a challenge and response process. The responses directly depend on the unique physical properties that is implicitly introduced during production. However, the uniqueness property is not sufficient to ensure security: the response to a particular challenge must also be unpredictable, i.e., it is required that the responses for a series of PUF challenges look like the output of a random function to an observer. PUFs have been proposed as tamper-evident building blocks for various use cases such as for authentication, key-derivation, and device identification [4, 9, 18].

So far many different PUF implementations in hardware have been proposed and evaluated over the recent years [1, 6, 18]. In particular, PUFs based on physical properties of memory cells in SRAM (Static Random Access Memory) and DRAM (Dynamic Random Access Memory) are very popular since they are cost-effective and widely deployed on many platforms. The declining costs of DRAM hardware further motivates recent research efforts on utilizing DRAM for PUFs, e.g., by measuring the decay rate [22, 23] or the inherent startup-value [20] of memory cells. In the following we focus on DRAM PUFs.

While DRAM hardware is highly standardized, cheap, widely used, and commercially available as off-the-shelf components, it also exhibits a number of reliability problems: Adverse conditions such as higher temperature or electromagnetic radiation can lead to bit errors in the information stored digitally on the hardware [10–12]. Moreover, researchers demonstrated that DRAM hardware is in fact subject to bit errors that are reproducable purely from software *under completely benign operating conditions*. In particular, several independent studies found that frequently accessing physically co-located memory cells leads to bit flips in adjacent memory cells [7, 8]. This effect, called *Rowhammer*, was subsequently shown to be exploitable by remote adversaries to undermine deployed security mechanisms on computing platforms using DRAM hardware [13, 17, 21]. The focus of the literature so far has been on Rowhammer-based attacks and defenses [2, 3, 7], however, recently, at HOST 2017 Schaller et al. considered a new paradigm to leverage the Rowhammer effect in DRAM to construct memory-based intrinsic Rowhammer PUFs (RH-PUFs) [16]. They propose to extract a unique device fingerprint from DRAM locations that are subject to reproducible bit flips.

Goals and Contributions. We revisit the security of DRAM PUFs and demonstrate two *remote* attacks on (Rowhammer) DRAM PUFs, proposed by Schaller et al. [16]. First, we show that an adversary observing few challenge-response pairs can predict future responses with high confidence, violating the unclonability property. Second, we present a Rowhammer-based Denial of Service (DoS) attack by tampering with the responses of DRAM PUFs. Our attacks work completely in software, i.e., without any physical access to the system. We implemented and evaluated our attacks in a real-world setup and in different scenarios. Our experimental results suggest that DRAM hardware might in fact not be a suitable candidate for secure PUF implementations. To summarize our contributions are:

- We present a remote, Rowhammer-based Denial-of-Service (DoS) attack against DRAM PUF constructions.

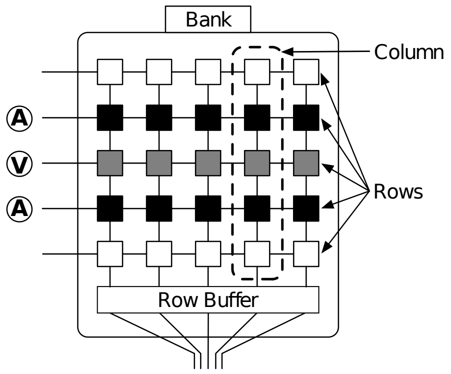


Figure 1: A DRAM bank is an organization of memory cells in a grid of columns and rows with a dedicated row buffer. *Rowhammer* exploits electro-magnetic coupling effects due to the increased chip density by activating Aggressor (A) rows that are physically adjacent to a Victim (V) row to influence its contents without accessing it.

- We present a modeling attack against the Rowhammer PUF proposed by Schaller et al. [16] that completely breaks its unclonability property.
- We implement and evaluate both attacks experimentally using a Pandaboard ES B3[1].

We would like to stress that our attacks are not limited to RH-PUF but concern DRAM PUFs in general. To this end, we discuss both of our attacks with respect to startup-value-based and decay-based DRAM PUFs in Section 6.

2 THE ROWHAMMER EFFECT IN DRAM

Modern DRAM hardware is structured hierarchically: the hardware unit itself is connected to the memory controller through one or multiple channels. The DRAM chips on the module consist of a number of *banks* which contain the memory cells. A bank is laid out in a grid of *columns* and *rows*, as shown in Figure 1.

DRAM rows contain neighboring memory cells, whereas columns hold adjacent cells. Each memory cell comprises a capacitor and a transistor, where the state of the capacitor is used to store an individual bit. Memory accesses always happen row-wise, i.e., read operations read out the cells of the entire row. The row data is retrieved and stored in a *row buffer*, before being transferred to the memory controller. However, read operations are destructive since retrieving bits from a row drains the capacitors of the corresponding memory cells. For this reason, a read operation also triggers a memory refresh of the current row by writing the content of the row buffer back to the drained cells of the current row.

The Rowhammer bug occurs during this write-back operation when accessing co-located rows of the same bank: frequently triggering (or *hammering*) an aggressor row (**A**) that is physically located on top of the victim row (**V**) affects the cells of the victim row due to electromagnetic coupling—although these cells are never accessed directly. Amplified by the shrinking feature size of the manifactured DRAM chips, several studies found Rowhammer to be a wide-spread reliability issue, and conclude that Rowhammer seems to be an inherent design problem of DRAM chips as all vendors and even ECC DRAM modules are affected [7, 8].

[1]https://svtronics.com/EVM/panda

3 DRAM PUFS & APPLICATIONS

RH-PUFs. While prior research on Rowhammer focused on its negative effect to illegitimately alter memory contents and break system's security mechanisms, Schaller et al. [16] investigated the properties of the Rowhammer effect in different DRAM chips as a PUF candidate. The RH-PUF consists of physically adjacent memory rows, where victim rows are located between aggressor rows. The RH-PUF has the following parameters: PUF_{addr} to indicate the starting address of the PUF in the memory, PUF_{size} refers to the number of *victim rows* and RH_{type} defines the order of victim rows and aggressor rows in a RH-PUF. RH_{type} can be either *double-sided*, referred to as DSRH, in which each victim row is located between two aggressor rows as shown in Figure 1, or *single-sided*, SSRH, in which two victim rows are located between two aggressor rows. These parameters can be used to define a RH-PUF instance in the DRAM memory such that $PUF_{RH} = PUF_{addr}, PUF_{size}, RH_{type}$. The challenge to the RH-PUF is the duration of the Rowhammer process RH_{time}. The idea of the RH-PUF, is to hammer the aggressor rows in the memory for the time defined by RH_{time} and then measure the changes in the victim rows. These changes, the indices of bit flips, constitute the RH-PUF response and are unique per memory chip. Thus, the response of a RH-PUF instance is a set of *integers*, i.e., indices of bit flips, rather than a binary value as in conventional delay-based PUFs [22]. As DRAM cell consists of a transistor and a capacitor, binary information is stored in a DRAM cell in the form of a charge on the capacitor. However, capacitors lose their charges gradually over time. Therefore, a periodic refresh is required to retain the charge stored in the capacitor, otherwise, the DRAM cell *decays* over time, i.e., loses its stored value. Note that periodic refresh of DRAM cells in the PUF region is disabled during RH_{time}. This allows for DRAM cells in the victim rows to decay over time, thus resulting in more bit flips. The RH-PUF falls under the decay-based DRAM PUFs category (more details in Section 7), therefore, it can be deployed in authentication protocols [23] as shown next.

Authentication Protocol. Xiong et al. [23] proposed to use the time-dependent decay characteristics of DRAM PUFs for authentication in a passive adversary model where the attacker has no physical access to the device. An important requirement or their DRAM PUFs is that for a series of PUF challenges the decay times RH_{time} must be strictly increasing. This is due to the fact that the security of the protocol relies on the number of *new* bit flips between the two PUF responses. Therefore, a challenge must be chosen such that guessing the locations of *new* bit flips is computationally infeasible, even if the attacker knows the previous responses [23]. When decay-based DRAM PUFs are used in authentication protocols, similarity of PUF response m with the reference response r stored in the verifier's database is measured using the Jaccard similarity J such that:

$$J(r, m) = \frac{|r \cap m|}{|r \cup m|} \tag{1}$$

The authentication protocol succeeds if $J(r, m) > J_{thresh}$, where J_{thresh} is defined for each PUF based on the noise of its measurements.

Key Derivation. Decay-based DRAM PUFs can be deployed in key derivation schemes [23]. A PUF response is typically transformed by a fuzzy extractor scheme from noisy non-uniformly distributed secret into a stable high-entropy key [5]. Fuzzy extractor corrects noisy PUF responses using error correcting codes and amplifies its randomness to generate a secret key. RH-PUF responses exhibit a maximum noise of 5%, which makes them suitable for key derivation schemes. Given the lower bound entropy of RH-PUF (0.192 per DRAM cell), a 128-bit key can be generated using a PUF_{size} of only 85 Bytes [16].

4 OUR ATTACKS ON DRAM PUFS

In this section we demonstrate our attacks on the RH-PUF [16]. Our attacks are feasible in the same adversary model as the RH-PUF, i.e., the memory region is exclusively allocated for PUF usage and cannot be accessed by other processes.

4.1 DoS Attack on the RH-PUF

As demonstrated in [16] cryptographic keys can be derived from the RH-PUF response caused by hammering PUF's (*aggressor rows*) for a fixed amount of time. As depicted in Figure 2 our DoS attack aims at maliciously modifying the PUF response (**V**) by hammering the borders of the RH-PUF region (**A**). Our attack is carried out without accessing the RH-PUF itself (black in Figure 2), i.e., by repeatedly accessing *physically adjacent* rows (white in Figure 2) above and below the PUF region, while simultaneously querying the PUF. Consequently, our malicious process can influence the RH-PUF response.

In principle, RH-PUFs can be queried at run time within the kernel module by allocating consecutive chunk of memory for the RH-PUF, disabling DRAM periodic refresh and selectively refreshing critical code sections while measuring RH-PUF response. For example, assuming a cryptographic key that is derived on-demand by RH-PUF is used at some point to encrypt sensitive data. Then, an adversary can perform this attack, when the PUF is queried again to decrypt the data. Related work already demonstrated how to exploit physical co-location in practice [13, 17, 21]: Since there is a non-linear physical-to-DRAM mapping between the physical addresses and the physical location in the DRAM module, a malicious process with knowledge of the mapping can trick the system's memory allocator into co-locating its memory adjacent to the PUF region. This attack results in a different key, as its seed is based on a faulty RH-PUF response, thus the user is prevented from retrieving the original data. This attack can also interfere with DRAM PUF-based authentication protocols causing the prover to be blocked from authenticating to the verifier. In Section 5.2, we present evaluation results of the DoS attack on several variants of the RH-PUF.

4.2 Modeling Attack on the RH-PUF

In [23], Xiong et al. demonstrated that DRAM-based PUFs can be queried at run time and utilized in authentication protocols in a passive adversary model, i.e., the attacker can only observe CRPs

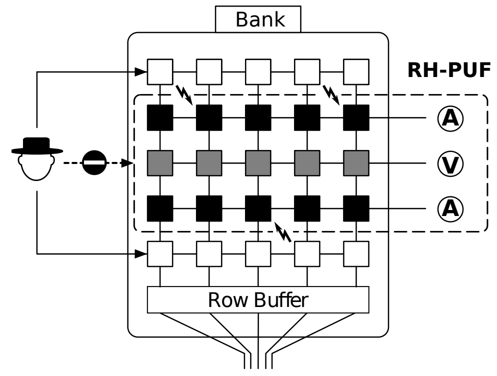


Figure 2: DoS Attack on the RH-PUF [16]: memory rows above and below the PUF region are accessed by the malicious process.

exchanged between the verifier and the prover. As explained in Section 3, challenges must adhere to certain criteria such that the probability of correctly guessing the responses is smaller than 2^{-128}. This implies that the difference in bit flips between two successive responses has to achieve a lower bound, denoted as ϵ_{bits}, to fulfill the security requirements. Therefore, if an adversary already knows the locations of bit flips in the previous response, it is computationally infeasible to guess the locations of the new bit flips bits in the following response, assuming that locations of the new bit flips have a uniform distribution. Modeling attacks of decay-based DRAM PUFs under these assumptions is challenging as the attacker has no access to PUF hardware and can observe but not chose CRPs, as in conventional modeling attacks on decay-based PUFs. However, a modeling attack is still feasible, as we show that the assumption of randomly-distributed bit flips does not hold true in practice. In particular, we observe that bit flips in two successive responses of RH-PUF are correlated. In other words, in one response, bits are more likely to flip near to bits that have already flipped in a previous response. Therefore, predicting the locations of next bit flips in the RH-PUF response is possible when enough CRPs are observed. Consequently, the RH-PUF can be modeled with reasonable probability. In Section 5.3, we present evaluation results of the modeling attack on several variants of RH-PUF.

5 EVALUATION

5.1 Setup

We implemented our attacks and evaluated the RH-PUF on a Pandaboard ES B3. The board has a TI OMAP 4460 System-on-Chip module with 1GB DDR2 memory, which is configured such that each memory row consists of 4KB. Our attacks are based on the open-source implementation of the RH-PUF [16] which is available online. In our experiments, we analyzed several RH-PUF instances with PUF_{size} = 4KB (1 row), 8KB (2 rows), 16KB (4 rows) and 32KB (8 rows) with RH_{type} set to single-sided (SSRH) or double-sided (DSRH). We analyzed the RH-PUF measurements using Matlab running on an Intel Core i7 desktop machine.

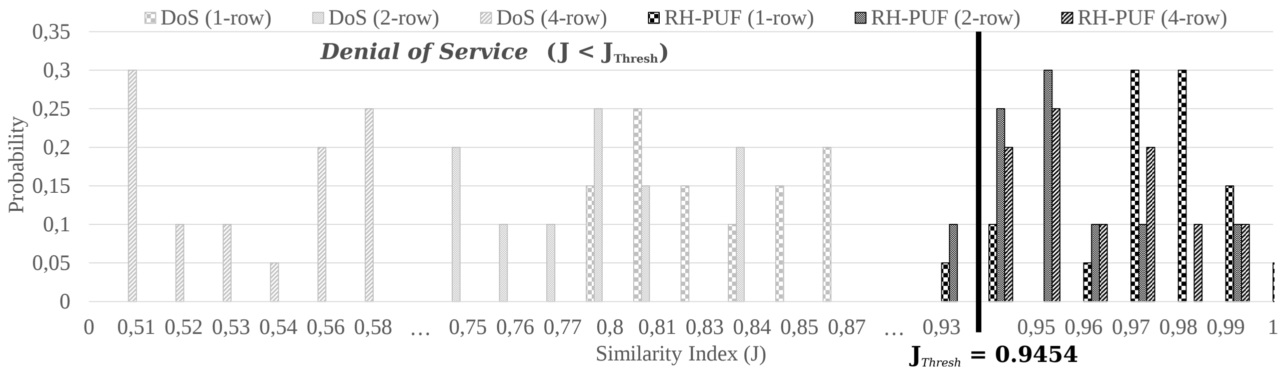


Figure 3: Histogram of J_{intra} values for three PUF instances before and after double-sided DoS attack using 20 measurements with PUF_{size} = 4KB, 8KB and 16KB, RH_{type}= SSRH and RH_{time} = 120s

5.2 DoS Attack on the RH-PUF

Our Denial of Service (DoS) attack can be launched remotely at run time by a malicious process that has simultaneous access to *physically adjacent* memory locations to RH-PUF memory. For our prototype implemention of the DoS attack we used the RH-PUF source code, which modifies the popular boot loader U-Boot [16]. We evaluated our attack on top of the RH-PUF, such that additional rows above and/or below the PUF region are hammered simultaneously while querying PUF responses. We present two variants of our DoS attack assuming that the malicious process has access to: i) one neighboring row above the PUF region, which we refer to as the *SSRH DoS attack* and ii) two neighboring rows above and below the PUF region, referred to as the *DSRH DoS attack* as shown in Figure 2. The goal of our attack is to increase the noise in the PUF measurements and reduce its robustness, such that the PUF can no longer be identified or used for key derivation.

To measure RH-PUF robustness, the Jaccard index J_{intra} is used to reflect similarity between two PUF responses for the same challenge (see Equation I). When $J_{intra} = 1$, it reflects that the responses are identical. As indicated in [16], RH-PUF achieves high robustness with a minimum of $J_{thresh} = 0.9454$ and exhibits a maximum noise of around 5% that can be corrected using standard fuzzy extractors in key derivation schemes [5]. We first measured responses of several RH-PUF instances, i.e., with different PUF_{size}, RH_{type} and RH_{time} values. Then, we performed the DoS attacks on these RH-PUF instances and collected their responses. In both experiments, each response was measured 20 times. Figure 3 shows the similarity index J_{intra} for the three RH-PUF responses before and after the attack. On the right-hand side of the figure, histograms of similarity index values J_{intra} for three SSRH RH-PUFs before the DoS attack are shown. Our results are on par with the results in [16]. On the left-hand side of Figure 3, the effect of our DoS attack on the J_{intra} values of the same three RH-PUF instances is shown. Our attacks shifted the similarity index of PUF responses J_{intra} away from the threshold value J_{thresh} and introduced a maximum of 80% noise to the PUF responses, which cannot be tolerated even with the use of more advanced fuzzy extractors. As indicated by the results in Table 1 and Table 2, our DoS attack works at best on RH-PUF with small sizes. For example, in the second row of Table 2, using DSRH DoS attack, the adversary is capable of reducing the J_{intra} of a DSRH RH-PUF of 4KB to 0.33, 0.20, and 0.56 (i.e., $\ll 0.9454 = J_{thresh}$ [16]) when hammered for 60s, 120s and 180s respectively. This is mainly due to the close proximity of the DoS aggressor rows to more PUF victim rows, since victim rows in the middle become less vulnerable to our attack when the RH-PUF size increases.

Table 1: SSRH DoS attack on several RH-PUF variants

RH_{type}	PUF_{size}	RH_{time}		
		60s	120s	180
SSRH	1-row PUF	0.5	0.78	0.853
SSRH	2-row PUF	0.79	0.85	0.76
SSRH	4-row PUF	0.46	0.68	0.88
DSRH	1-row PUF	0.67	0.53	0.69
DSRH	2-row PUF	0.89	0.9	0.88
DSRH	4-row PUF	0.89	0.97	0.97

Table 2: DSRH DoS attack on several RH-PUF variants

RH_{type}	PUF_{size}	RH_{time}		
		60s	120s	180
SSRH	1-row PUF	0.6	0.84	0.88
SSRH	2-row PUF	0.33	0.79	0.85
SSRH	4-row PUF	0.37	0.54	0.73
DSRH	1-row PUF	0.33	0.2	0.56
DSRH	2-row PUF	0.85	0.85	0.92
DSRH	4-row PUF	0.94	0.96	0.88

5.3 Modeling Attack on the RH-PUF

Our modeling attack on the RH-PUF works under the restriction that an attacker can *only* observe CRPs exchanged in the authentication protocol. Hence, we perform the following steps: i) measure RH-PUF CRPs for a RH-PUF instance, ii) pick *valid* CRPs that adhere to the criteria defined in [23] as training set from the collected CRPs, and iii) run an interpolation algorithm to predict future responses based on the training set and verify its accuracy. We use standard

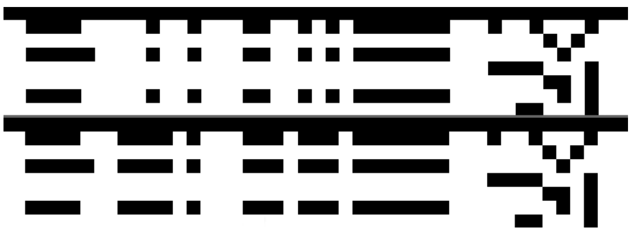


Figure 4: Visual representation of the evolution of bit flips in a bank (charged cells are depicted in black, uncharged cells in white pixels). The first measurement is displayed on top, the second measurement on the bottom.

interpolation algorithms for two reasons: we aim to show that the attacker does not require to craft a dedicated algorithm to perform the attack and that indeed publicly available interpolation algorithms can be utilized to easily launch our attack.

The basic idea is best explained in Figure 4: it depicts two subsequent measurements of the same location in DRAM. Black pixels represent memory cells that are set to 1, white pixels represent cells that are set to 0. We made two observations, (1) many bit flips stabilize over time, and (2) additional bit flips tend to appear around existing clusters. We tested several interpolation methods that are usually used for the manipulation of digital imagery. Among them *Piecewise Cubic Hermite Interpolating Polynomial* (PCHIP) achieved the highest prediction rates. Therefore, we use the PCHIP algorithm to predict future bit flip locations by interpreting the memory contents obtained in our measurements as binary images. For each RH-PUF instance, we first measured 335 CRPs with $2s$ steps between each two challenges in the range of $30s$ to $700s$. We observed that within this time range the total number of bit flips in each response are less than 2KB. Then, for each valid challenge, we calculated ϵ_{bits} based on the number of bit flips in its response to choose the *next* valid challenge from the available CRPs, such that the probability of correctly guessing the next response is smaller than 2^{-128}. For the same RH-PUF instance, ϵ_{bits} values are different based on the value of the authentication threshold J_{thresh}, which defines if a measured response is accepted based on its similarity to a reference response, i.e., $J_{intra} > J_{thresh}$. We observed that the smaller this authentication threshold is, the larger ϵ_{bits} values become. [2] After collecting valid CRPs in this way, we used them as training set for our modeling algorithm. For modeling the RH-PUF behavior, we used an interpolation algorithm in Matlab. We first interpolate CRPs in between two valid CRPs. While these interpolated CRPs are not used for authentication, they allow our algorithm to identify bit flip cluster regions in the DRAM. We then extrapolate from these values by predicting future responses with high confidence as shown in Table 3. In Table 3, each row shows the minimum number of required CRPs based on the authentication threshold J_{thresh} and RH_{type} to achieve a prediction rate that is higher than J_{thresh}.

6 DISCUSSION

As our evaluation results show both attacks succeed and completely undermine the security guarantees of the RH-PUF construction.

[2]Due to space limit we refer the reader to [23] for more details about ϵ_{bits} calculations.

Table 3: Modeling 8-row RH-PUFs

RH_{type}	Required CRPs	J_{thresh}	Prediction rate
DSRH	10	0.6	0.75
DSRH	11	0.7	0.8
DSRH	14	0.8	0.88
SSRH	12	0.6	0.73
SSRH	14	0.7	0.79
SSRH	17	0.8	0.83

Furthermore, our attacks are practical and work completely from software, i.e., assuming only a remote adversary. There are two main consequences: first, the DRAM-based RH-PUF responses are subject to bit flips by accessing memory that is physically co-located. This is possible even under a remote adversary model, since the attacker can force physical co-location of her memory to the PUF region. While previous defensive work demonstrated how to prevent user-space attackers from co-locating memory with kernel-space memory [3], adopting such a physical isolation policy in the context of PUFs is difficult, as the memory of the PUF region might be managed by a different allocator than the attacker's memory. Over-allocating system memory for the PUF region such that a large number of rows surrounds the actual PUF region might be a possible strategy. However, this comes at the cost of blocking a large amount of memory resources and will affect the performance of the system. Another way could be to prevent other processes from running on the device while measuring the PUF response. However, this effectively locks the device during PUF usage. The second consequence is that the RH-PUF construction in its current design is predictable, as our modeling attack succeeds with high confidence. The reason is that vulnerable DRAM cells are not distributed uniformly but contrarily tend to form clusters on the module. Hence, correlating bit flip locations physically is feasible. Since this is due to the intrinsic properties of DRAM hardware, all DRAM-based PUF implemenations assuming uniform distribution (in time or space) of decay, startup value, or bit flips are affected and must be considered inherently insecure.

7 RELATED WORK

Recently, research work showed that a PUF-behavior can be found in DRAM memory [15]. Since then, several DRAM-based PUFs have been proposed and evaluated. They differ in which DRAM characteristics they exploit and their accessibility. In the following, we go briefly over the different types of DRAM-based PUFs.

Start-up DRAM PUFs. Similar to SRAM memory, DRAM memory exihibts a unique behavior that relies on the startup tendencies of DRAM cells [19]. When powering up DRAM, the capacitors in DRAM cells show different inclinations towards either a 0 or a 1 due to uncontrollable process variations. Hence, the startup values of DRAM cells can be used as a unique fingerprint. However, in practice, it can only be evaluated at system power up.

Decay-based DRAM PUFs. As capacitors lose their charges over time, which is referred to as decay time, DRAM cells require periodic refresh in order to retain their values. The DRAM memory controller performs the periodic refresh at a vendor-defined rate, which is usually 32ms or 64ms. If periodic refresh is disabled, then, some DRAM cells decay to 0, while others decay to 1. Exploiting the unique decay rate of DRAM cells to construct a unique identity has also been proposed for identification and key storage in FPGAs [14] and in comoditiy devices at run time [23]. Extracting Decay-based PUF response implies disabling the refresh cycle of DRAM cells. However, disabling periodic refresh for the whole memory can lead to system crash, while partially disabling periodic refresh for a specified memory region is not possible. Therefore, several techniques were proposed to cope with this issue. One solution is to perform a selective DRAM refresh. In which periodic refresh is disabled and a read-loop is used to refresh the contents of the whole memory except the PUF region.

8 CONCLUSION

Recent research proposed to leverage DRAM hardware for the construction of PUFs. Our findings show that DRAM-based PUF implementations are subject to remote attacks, such as Denial of Service. Futhermore, a passive adversary can predict future DRAM PUF responses by observing only few challenge-response pairs. Our results show that DRAM hardware may in fact not be a suitable candidate for the construction of secure PUF implementations.

Acknowledgments. This work was supported by the German Science Foundation CRC 1119 CROSSING P3, the German Federal Ministry of Education and Research (BMBF) in the context of CRISP (IP2), and the Intel Collaborative Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS).

REFERENCES

[1] F. Armknecht, R. Maes, A.-R. Sadeghi, B. Sunar, and P. Tuyls. Memory leakage-resilient encryption based on physically unclonable functions. In *Towards Hardware-Intrinsic Security*. Springer, 2010.

[2] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. Anvil: Software-based protection against next-generation rowhammer attacks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016.

[3] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi. Can't touch this: Software-only mitigation against rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*, 2017.

[4] G. Clarke, D. Van Dijk, and S. Devadas. Controlled physical random functions. *Annual Computer Security Applications Conference (ACSAC)*, 2002.

[5] J. Delvaux, D. Gu, I. Verbauwhede, M. Hiller, and M.-D. M. Yu. Efficient fuzzy extraction of puf-induced secrets: Theory and applications. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2016.

[6] S. Katzenbeisser, Ü. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann. Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon. *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2012.

[7] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *International Symposium on Computer Architecture (ISCA)*. ACM/IEEE, 2014.

[8] M. Lanteigne. How rowhammer could be used to exploit weaknesses in computer hardware. https://www.thirdio.com/rowhammer.pdf, 2016.

[9] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. Van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication applications. In *VLSI Circuits, Digest of Technical Papers*. IEEE, 2004.

[10] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms. In *ACM SIGARCH Computer Architecture News*, volume 41. ACM, 2013.

[11] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *International Conference on Dependable Systems and Networks (DSN)*. IEEE/IFIP, 2015.

[12] N. Nikiforakis, S. Van Acker, W. Meert, L. Desmet, F. Piessens, and W. Joosen. Bitsquatting: Exploiting bit-flips for fun, or profit? In *International Conference on World Wide Web (WWW)*. ACM, 2013.

[13] R. Qiao and M. Seaborn. A new approach for rowhammer attacks. In *International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2016.

[14] A. Rahmati, M. Hicks, D. E. Holcomb, and K. Fu. Probable cause: the deanonymizing effects of approximate dram. *ACM SIGARCH Computer Architecture News*, 43(3), 2016.

[15] S. Rosenblatt, S. Chellappa, A. Cestero, N. Robson, T. Kirihata, and S. S. Iyer. A self-authenticating chip architecture using an intrinsic fingerprint of embedded dram. *IEEE Journal of Solid-State Circuits*, 48(11), 2013.

[16] A. Schaller, W. Xiong, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer. Intrinsic rowhammer pufs: Leveraging the rowhammer effect for improved security. In *International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017.

[17] M. Seaborn and T. Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.de/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2016.

[18] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Design Automation Conference (DAC)*. ACM/IEEE, 2007.

[19] F. Tehranipoor, N. Karimian, K. Xiao, and J. Chandy. Dram based intrinsic physical unclonable functions for system level security. In *Great Lakes Symposium on VLSI*. ACM, 2015.

[20] F. Tehranipoor, N. Karimian, W. Yan, and J. A. Chandy. Dram-based intrinsic physically unclonable functions for system-level security and authentication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(3), 2017.

[21] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. Drammer: Deterministic rowhammer attacks on commodity mobile platforms. In *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[22] W. Xiong, A. Schaller, N. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer. Practical dram pufs in commodity devices. *IACR Cryptology ePrint Archive*, 2016.

[23] W. Xiong, A. Schaller, N. A. Anagnostopoulos, M. U. Saleem, S. Gabmeyer, S. Katzenbeisser, and J. Szefer. Run-time accessible dram pufs in commodity devices. In *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2016.

C ON THE SECURITY OF STRONG MEMRISTOR-BASED PHYSICALLY UNCLONABLE FUNCTIONS

[3] Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni, Ahmad-Reza Sadeghi. "On the Security of Strong Memristor-based Physically Unclonable Functions". In Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC'20), 2020.

On the Security of Strong Memristor-based Physically Unclonable Functions

Shaza Zeitouni, Emmanuel Stapf, Hossein Fereidooni and Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{shaza.zeitouni, emmanuel.stapf, hossein.fereidooni, ahmad.sadeghi}@trust.tu-darmstadt.de

***Abstract*—PUFs are cost-effective security primitives that extract unique identifiers from integrated circuits. However, since their introduction, PUFs have been subject to modeling attacks based on machine learning. Recently, researchers explored emerging nano-electronic technologies, e.g., memristors, to construct hybrid-PUFs, which outperform CMOS-only PUFs and are claimed to be more resilient to modeling attacks. However, since such PUF designs are not open-source, the security claims remain dubious. In this paper, we reproduce a set of memristor-PUFs and extensively evaluate their unpredictability property. By leveraging state-of-the-art machine learning algorithms, we show that it is feasible to successfully model memristor-PUFs with high prediction rates of 98%. Even incorporating XOR gates, to further strengthen PUFs' against modeling attacks, has a negligible effect.**

***Index Terms*—memristor PUFs, machine-learning, modeling attacks.**

I. Introduction

Physically unclonable functions (PUFs) are considered as an innovative and cost-effective replacement for secure non-volatile memory, e.g., EEPROMs, or expensive cryptographic hardware engines to realize light-weight authentication as well as key generation and storage [1]. Silicon-based PUFs leverage the uncontrollable manufacturing process variation of integrated circuits (ICs) as a source of entropy to derive a device-specific cryptographic key or unique identifier. Hence, a PUF can be expressed as a black-box system that takes a *challenge* as an input and returns a unique output, also called a *response*.

PUF classifications. Based on their input-output space, PUFs are roughly classified into two broad categories, *weak* and *strong* PUFs [1]. Weak PUFs can only process a limited number of unique challenges and are mainly deployed in cryptographic key generation schemes. On the other hand, strong PUFs possess a large number of challenges such that, within a bounded amount of time, it is infeasible to measure all challenge-response pairs (CRPs), making them a perfect primitive for authentication protocols [1], [2]. The security of PUF-based protocols/schemes relies on the security of the underlying PUFs. Therefore, some properties should be satisfied in PUFs: 1) uniqueness, the behavior of a PUF instance should be unique among other instances of the same design, 2) unclonability, meaning that even a PUF manufacturer cannot create two indistinguishable or identical PUF instances, 3) unpredictability, PUF responses to unseen challenges should be unpredictable even if some CRPs are known, and 4) robustness or reliability, a PUF response to the same challenge should be steady under different operating conditions.

PUFs under attack. Since their introduction, PUFs have been heavily under attack whether to reveal PUF-based secret keys or to emulate their behavior. The attacks range from physical attacks, which can be further categorized into invasive/semi-invasive [3], [4] and side-channel attacks [5], to software-based modeling attacks where no physical access to the PUF is needed [6]. Software attacks can be launched remotely and are cheaper than physical attacks that require expensive tools. Therefore, software attacks can scale to a large number of devices. Software-based modeling attacks leverage machine learning (ML) algorithms to learn an accurate model of the PUF under attack using a set of measured CRPs. Modeling attacks mainly target strong PUFs with a large number of CRPs. The PUF model generated by a ML algorithm is said to be accurate, if it predicts responses of a PUF to unseen challenges with high probability, thereby undermining the unpredictability property of strong PUFs. Subsequently, the security of protocols building on top of modeled PUFs is broken. During the last decade, novel and improved PUF designs have been introduced but were later successfully broken by modeling attacks [6], [7].

Deploying emerging technologies to secure PUFs. Due to prominent and continuous progress in material science, novel *nano-devices* have been developed for beyond-silicon applications. Examples of such nano-devices include carbon nanotube field-effect transistors [8], memristors [9], spintronic-logic devices [10] and many others. Being compatible with CMOS manufacturing technology motivated the deployment of such technologies in the fabrication of digital circuits, e.g., non-volatile memory (NVM) architectures, such as resistive random access memory (RRAM) and magneto-resistive random access memory (MRAM). Such hybrid memory architectures offer great advantages over CMOS-only technology due to their higher density, lower footprint and lower power consumption. As memristive devices have a stochastic nature when switching their resistance [9], [11], in addition to their sensitivity to process variation, they have been seen as an inherent source of entropy that can be leveraged to build PUFs. Consequently, several so-called *hybrid* PUF designs have been proposed, including strong PUFs [12]–[14] as well as weak PUFs [15]. Most newly proposed strong PUFs (e.g., [13], [14], [16]) are not thoroughly tested against most recent machine learning algorithms, e.g., deep neural networks that are successfully deployed to model complex functions in other fields of research [17]. Nevertheless, they are claimed to be resilient to modeling attacks.

Challenges. Hybrid PUF designs are mainly evaluated with circuit simulators, since such nano-technologies, e.g. memristors, are not industry-ready yet. In simulation, the PUF circuit and its components, e.g., transistors, memristors, etc., are described using mathematical functions. When a challenge is fed as an input to the PUF circuit, the circuit simulator performs a step-wise evaluation of the mathematical functions describing the PUF circuit to compute its response. While many new hybrid PUF designs have been proposed, verifying their properties and security guarantees is highly challenging. One reason is that only the high-level description of the PUF designs and the resulting security-relevant characteristics are published. However, neither the resulting datasets (CRPs) nor the low-level details, e.g., the parameters' values of the components, which are required to reproduce the PUF's intended behavior or CRPs, are published. The lack of open specifications makes it difficult to verify the claimed characteristics and resilience to machine learning attacks.

Our goals and contributions. The main motivation of this work is to evaluate the security of hybrid memristor PUFs that are claimed to be resilient to modeling attacks. Our contributions are as follows:

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

- We reproduce a set of memristor-based PUFs [14], [16], [18] and simulate their behavior to collect CRPs. We then verify their resiliency to modeling attacks using the measured CRPs only, i.e., as a black-box system. We achieve accuracy rates higher than 95% using state-of-the-art ML algorithms and thus, break the security guarantees of the tested PUFs.
- We further evaluate XOR-based versions of the PUFs under test, since more complex behavior could potentially result in stronger PUFs. Subsequently, we show that even the XOR-based versions can be successfully modeled with higher computational power.
- We evaluate the effect of noise on the learnability of the PUFs. Our results show that even when noisy responses are used for model training, it is still feasible to obtain a model with a prediction accuracy higher than 90%.

II. Background: Physical Unclonable Functions

Silicon-based PUFs are noisy electronic circuits that are stimulated with a challenge in order to produce a response. A PUF response strongly depends on both the provided input and the innate physical properties of the integrated circuit embedding the PUF. Based on PUF's underlying components and architecture, three major groups of PUFs can be identified; *delay-based* PUFs, *memory-based* PUFs and *hybrid* PUFs.

Memory-based PUFs, e.g., SRAM-based PUFs, which leverage variations in the manufacturing process that lead to a mismatch of transistors among SRAM memory cells [19]. Extracting device fingerprints of non-volatile memory technologies were also introduced in [15], [20]. Memory-based PUFs, however, can only produce a limited number of responses, linear in the number of utilized memory cells. Therefore, they are considered as *weak* PUFs and mostly deployed in cryptographic key generation schemes [1].

Delay-based PUFs leverage delay difference between two identical signal-propagation paths. The delay difference is caused by wires and transistors mismatch, due to the process variation, in identical paths. Two prominent examples of delay-based PUFs are *ring-oscillator* PUFs (RO-PUFs) and *arbiter* PUFs (APUFs). Several APUF variants, e.g., the XOR-APUF [21] and the Feed Forward APUF [22], have been proposed to strengthen APUFs against modeling attacks. However, all APUF variants have been subject to successful modeling attacks [6], [7], [23]. An APUF can generate an exponential number of responses, 2^n, with respect to the number of switches n, which are the main components of an APUF. Therefore, it is mainly deployed in authentication protocols [1], [2].

Hybrid PUFs leverage the recently-emerging nano-devices, which can be combined with conventional delay-based PUFs [12]–[14], [24] promising great advantages (higher robustness and resilience to modeling attacks) over CMOS-only PUFs due to the stochastic behavior of such nano-devices. However, verifying their security is challenging, since the proposed PUF designs are not publicly available. Thus, making it harder for the research community to verify their claimed characteristics. As a result, many newly proposed PUFs are not thoroughly tested against state-of-the-art machine learning.

III. Adversary Model

Our adversary model adheres to the established assumptions of strong PUFs and modeling attacks [6], [7], [25]. The adversary is able to collect a large number of challenge-response pairs (CRPs) in plain text. Since this work deals with simulated PUF designs, we do not consider/rely on any sort of physical attacks, e.g., side-channel analysis, to support ML-based modeling attacks [26]. Moreover, we assume a black-box model, i.e., our adversary receives only CRPs of the PUF under attack, no further information are obtained, e.g., the mathematical model representing the PUF behavior. In our evaluation in § VI-B, we differentiate between two types of adversaries. A *weak* adversary with access to limited computational power, e.g., in form of an off-the-shelf laptop, and an *advanced* adversary that has access to more powerful computational resources with specialized hardware to execute ML algorithms such as a Tensorflow laptop.

IV. Tested Memristor-based PUFs

Memristor Arbiter PUF [16], shown in Fig. 1, has a fairly simple design compared to the classical APUF. It consists of two identical chains of memristors connected in series, which are connected to the input ports of a D Flip-Flop (D-FF) to define the response. Each memristor is connected in parallel to an NMOS transistor that acts as a switch. The gate of each transistor is controlled by one challenge bit. When a challenge bit is 0, the transistor is turned off and the corresponding memristor is included in the signal propagation path, otherwise, the signal propagates through the transistor itself due to its small R_{on} resistance. The workflow for the Memristor Arbiter PUF is as follows: first, in the *reset phase* V_{rst} and $\overline{V_{rst}}$ are enabled, while all switch-transistors are turned off, such that the memristors are driven to random resistance states depending on their individual properties obtained due to their stochastic switching behavior and process variation. Then, in the *challenge application phase*, a pulse is fired at the beginning of both memristor chains and the voltages at the gates of all switch-transistors are applied simultaneously based on challenge bits. In this phase, V_{ctrl} is also enabled to prevent voltage levels at the inputs of the D-FF to rise. During the *response evaluation phase*, V_{ctrl} is disabled, allowing the signal to propagate in both chains and be captured at the D-FF inputs to generate a response.

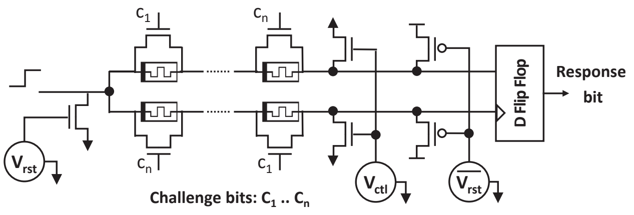


Fig. 1. Memristor Arbiter PUF Design Proposed in [16].

RRAM Arbiter PUF [14], shown in Fig. 2 consists of two identical chains of 1 transistor - 1 resistive device (1T1R) cells and switches (similar to classical APUF) connected in series through relays. Such that two 1T1R cells and a switch comprise one PUF stage. Relays are used to disconnect 1T1R cells from the switches during the reset of memristors. The RRAM Arbiter PUF works as follows: all 1T1R cells are first reset to random high resistance states (HRS). To query the PUF, the relays are switched on to connect 1T1R cells to the switches, which are controlled by challenge bits. A read pulse of 0.1 v, which is applied at the BL lines of 1T1R cells of the first PUF stage, propagates through the two established paths and is captured at the inputs of a voltage comparator to generate a response. The RRAM Arbiter PUF design is more complex than Memristor Arbiter PUF design and is, as our results in § VI-B show, harder to break.

Memristor RO-PUF (mrPUF) [18], shown in Fig. 3, consists of a nanoscale $N \times M$ crossbar array of memristors connected to two identical Current-Mirror Ring Oscillators (CM-ROs). The rest of mrPUF components are similar to a conventional RO-PUF; two N-to-1 analog multiplexers, two counters and a comparator to capture

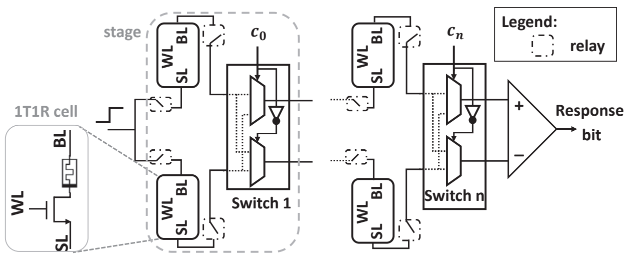


Fig. 2. RRAM Arbiter PUF Design Proposed in [14].

the response. The idea of mrPUF is to translate memristors' states into frequencies through ROs. To query the mrPUF, addresses of two memristors, i.e., a column number and two row numbers are specified. Although a mrPUF can only generate a limited number of CRPs, the authors proposed to deploy it in authentication protocols. In order to achieve a higher number of CRPs and a more complex behavior we also build an XOR-based mrPUF as we show in § VI-A.

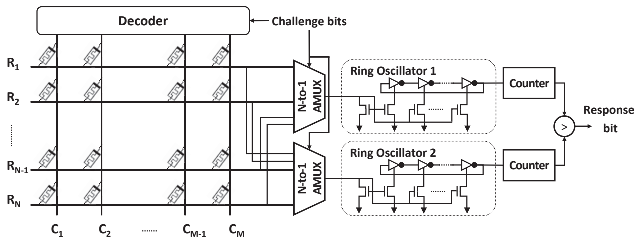


Fig. 3. mrPUF Design Proposed in [18].

V. Deployed Machine Learning Algorithms

We evaluate the unpredictability of the aforementioned memristor-PUFs in § IV against several machine learning algorithms. We describe next a subset of the implemented machine learning algorithms and elaborate why we use them during our evaluation in § VI-B.

Logistic Regression (LR) is a classification algorithm that was heavily used earlier to attack the classical arbiter and XOR arbiter PUFs [6], [25]. In newer strong PUF designs, LR was used to show the resilience of the design against ML-based modeling attacks [16]. We use LR in our attacks to verify the results of the tested PUFs.

Ensemble classifiers (meta-algorithms) are a type of classifiers, which were used to attack newer strong PUF designs that could not be broken by LR [25]. We use ensemble classifiers to verify if they can also be used to model memristor-PUFs. In ensemble algorithms, a number of weak classifiers are combined (through averaging or max vote) to create a model that can make accurate predictions. We use the ensemble classifiers Random Forest and AdaBoost. Random Forest [27] is an improvement over bagged Decision Trees. It fits a number of Decision Tree classifiers on various sub-samples of the training dataset and uses averaging to improve the prediction accuracy and to control over-fitting. AdaBoost [28] is a boosting algorithm with the *Decision Tree* classifier as the base estimator. It builds a model from the training data, then creates another model that attempts to correct the errors of the previous model. Models are added until the training set is predicted correctly. During the learning phase, data, which is misclassified by the previous learners, is given more weight by the subsequent learners.

Recurrent Neural Networks (RNN) belong to the group of artificial neural networks. RNNs have been successfully applied in different research fields, e.g. speech recognition, to learn complex models [17]. RNNs are state-of-the-art ML algorithms designed to recognize patterns in data sequences. This makes RNNs perfectly suited for ML-based PUF modeling, since PUF's challenges can be represented as binary sequences. Therefore, they are strong candidates to model PUF designs that could not be learned with simpler algorithms. An RNN has a looping mechanism that allows information to flow from one step (in sequence) to the next. This information is the hidden state, which is a representation of previous inputs. However, RNNs suffer short-term memory. In most real-world problems, a variant of RNN such as Gated Recurrent Unit (GRU) [29] is used to solve the problem of short-term memory using a mechanism called *gates*, which learn what information to add or to remove from the hidden state. Thus, gates enable learning long-term dependencies.

VI. Evaluation

A. PUF Parameters

The tested PUF designs are neither completely open-source, due to unavailable memristor models or missing parameters' values of some components, nor their resulting CRPs are disclosed for research. Therefore, to collect CRPs for modeling attacks, we reproduced the PUFs using open-source memristor models that exhibit similar behavior to the ones deployed in the original designs. We also tuned the missing parameters' values in PUF circuits in order to obtain security properties (uniqueness and uniformity[1]) close or similar to the values achieved by the original designs. We performed PUFs simulations using Cadence Spectre on a CentOS 7 Desktop with an Intel Core i5-2400 CPU running at 3.10 GHz and 16 GB RAM.

Memristor Arbiter PUF [16]. We build Memristor Arbiter PUF in several challenge sizes, i.e., number of memristors in chain, 64-bit and 128-bit. For our simulations, we utilize the available models and parameters as suggested in [16]. In particular, we used the compact model by Rák et al. [30] for memristors with the parameter values: $D = 10$nm, low resistance state (LRS) or $R_{on} = 121\Omega$, high resistance state (HRS) or $R_{off} = 12.1K\Omega$ and $\mu = 10^{-14}m^2(V.s)$. A process variation of 15% is applied to D as suggested by the authors [16]. For CMOS transistors, we use the PTM device model for 45nm CMOS technology and set process variation to 15% for threshold voltages of NMOS and PMOS transistors. Table I shows the statistics of the simulated Memristor Arbiter PUF with tuned transistors parameters, channel length $L = 45$nm and channel width $W = 45$nm (except for NMOS transistors connected to the V_{ctrl}, which have a channel width W between 1μm and 3μm).

TABLE I
Memristor Arbiter PUF Statistics & Simulation Times

PUF size	Uniqueness	Uniformity	Simulation Time per CRP
64 bits	50.6%	48.3%	44.5 s
128 bits	52.8%	52.4%	104.2 s

RRAM Arbiter PUF [14]. We build and simulate the PUF in two sizes, 16-bit and 24-bit, as given in [14]. We utilize the PTM model for 65nm CMOS transistors, which are used for access transistors and transistors within the analog multiplexers and inverters of the switches, and set process variation to 10% for threshold voltages of NMOS and PMOS transistors as suggested by the authors. However, for memristors, since the model used in the paper was not publicly available at the time of experiments, we utilize an open-source

[1]It refers to the proportion of 0's and 1's in responses of different PUF instances to a given challenge

Verilog-A model targeting the same $TiN/HfO_x/TiN$ RRAMs, provided by Chen et al. [11] with a process variation of 30%, 10% and 10% for the parameters $I0$, $gamma0$ and $Vel0$ as suggested in [11]. Table II shows the statistics for the simulated RRAM Arbiter PUF with transistors of channel length $L = 70$nm and channel width $W = 70$nm for NMOS and $W = 0.7\mu$m for PMOS transistors.

TABLE II
RRAM ARBITER PUF STATISTICS & SIMULATION TIMES

PUF size	Uniqueness	Uniformity	Simulation Time per CRP
16−bit	47.2%	49.5%	246 s
24−bit	49.5%	50.1%	388 s

Memristor RO-PUF (mrPUF) [18]. We implement the mrPUF in three sizes, 40, 64, and 128 rows. We utilize the same components and models as described in [18]. For transistors we use Cadence's generic process design kit (GPDK) model for 90nm CMOS technology and apply the same process variation suggested by Cadence. For memristors we utilize the Verilog-A model by TEAM [31] and set LRS to 500KΩ and HRS to 200MΩ with process variation of 10% for both resistance values. Table III shows the statistics for the given parameter values. Uniqueness measurements are performed among columns in the same crossbar array.

TABLE III
MRPUF STATISTICS & SIMULATION TIMES, PUF SIZE IS GIVEN AS $N \times M$, N: NUMBER OF ROWS, M: NUMBER OF COLUMNS

PUF size	Uniqueness	Uniformity	Simulation Time per CRP
40×4	54.9%	50.3%	21 s
64×4	54.5%	49.9%	37 s
128×4	55.2%	50.7%	55 s

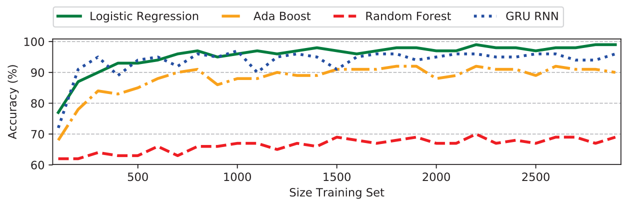
B. Modeling Attacks Using Machine Learning Algorithms

We performed modeling attacks with several ML algorithms: Logistic Regression, Support Vector Machine, voting ensembles, stochastic gradient boosting, AdaBoost, Extra Trees, Bagged Decision Trees, XGBoost, Random Forest and RNNs. We implemented the aforementioned algorithms using Python libraries Scikit-learn and Keras, which provides an interface to Tensorflow. However, we only show the results obtained by Logistic Regression, Random Forest, AdaBoost and GRU RNNs due to the limited space. CRPs generated by simulating the PUFs under test are presented in form of a matrix, where each row contains the challenge and response bits in **binary** form. In Table IV, we summarize the ML parameters' values used for training models of all reproduced PUF designs. These parameters can be further tuned for each PUF to achieve higher prediction rates, however, we used unified parameters values for the sake of comparison among the tested PUFs.

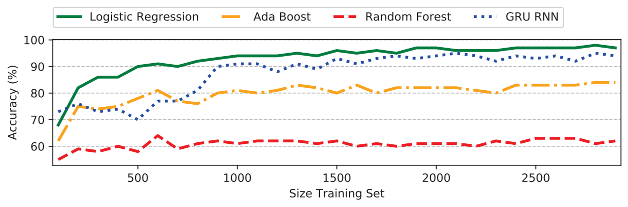
TABLE IV
PARAMETERS OF USED MACHINE LEARNING ALGORITHMS

Classifier	Hyper-parameters
$Random\ Forest$	$estimators = 100$
$AdaBoost$	$estimators = 50,\ learning\ rate = 1.0$
LR	$C = 1.0,\ penalty = l2$
RNN	$Layers = [GRU, GRU, GRU, Dense]$ $Layer\ size = [128, 128, 128, 1]$ $Optimizer = Adam$ $Loss = binary\ crossentropy$ $epoch = 100,\ learning\ rate = 0.001$

Memristor Arbiter PUF. The results of our modeling attacks are shown in Fig. 4. The best prediction results are achieved with Logistic Regression and GRU RNN, which achieved 99% and 96% accuracy for the 64-bit PUF and 98% and 95% accuracy for the 128-bit PUF, respectively. This indicates a linear relation ship between the challenges and the corresponding responses. We believe that this could be due to the simple design of Memristor Arbiter PUF, e.g., memristors are either included in the propagation paths or not, there is no switching between propagation paths as in the classical arbiter PUF. The AdaBoost achieves a prediction accuracy of 92% for the 64-bit PUF and 84% for the 128-bit PUF, respectively. Whereas, Random Forest achieves only 70% accuracy for the 64-bit PUF and 63% accuracy for the 128-bit PUF. Although the resulting statistical values of our simulated Memristor Arbiter PUF are close to the original values [16], our modeling results, however, do not correspond to the results presented in [16]. The authors claim a resilience of the Memristor Arbiter PUF against modeling attacks and present results for Linear Regression with only 50% prediction accuracy. Nevertheless, we achieved accuracies of over 95%. This shows the importance of open-sourcing proposed PUF designs. Based on our results, we cannot confirm the security guarantees claimed in [16].



(a) 64-bit Memristor Arbiter PUF.

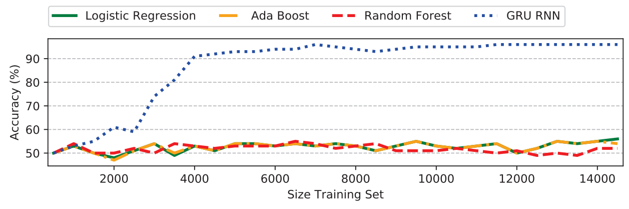


(b) 128-bit Memristor Arbiter PUF.

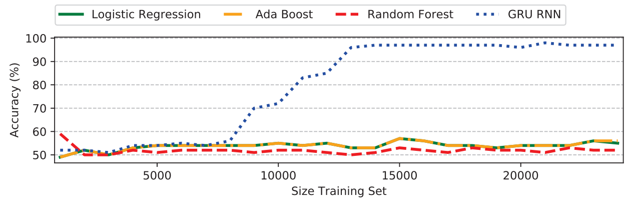
Fig. 4. Prediction Accuracy on Memristor Arbiter PUF.

RRAM Arbiter PUF. In Fig. 5, the prediction accuracy of our modeling attacks is shown for two PUF sizes, 16-bit and 24-bit. For both sizes, only the multi-layered neural networks (GRU RNN) is able to break RRAM Arbiter PUF. For breaking the PUF, a training set size of around $7,000$ CRPs for the 16-bit PUF and of around $13,000$ CRPs for the 24-bit PUF is needed to achieve an accuracy of more than 95%. For the test set, we use $1,000$ CRPs. All other algorithms only achieve prediction accuracies around 50% which resembles guessing. In Fig. 6, the duration for learning an accurate model of RRAM Arbiter PUF is shown. The plots show that the learning time only increases linearly with the number of CRPs and the PUF size. Therefore, we believe that also larger PUF sizes can be modeled with higher computational resources and/or longer learning durations. However, in our experiments we only tested up to 24-bit, due to the long simulation time of RRAM Arbiter PUF that is required to collect enough CRPs. As indicated in Table II, a larger PUF requires longer simulation time. For the 24-bit PUF, learning the model takes $1,728$ s for $13,000$ CRPs, which means that each one of the epochs of the GRU RNN takes 24.7 s on average. All other algorithms have a negligible model learning time of less than 1 s, which is why the lines for Logistic Regression, AdaBoost and Random Forest form one solid line in the plots of Fig. 6.

Modeling attacks on both Memristor Arbiter PUF and RRAM Arbiter PUF were performed on an off-the-shelf laptop, a T440s ThinkPad with Intel i7-4600U CPU and 12 GB RAM, meaning that even a weak attacker can break the PUFs.

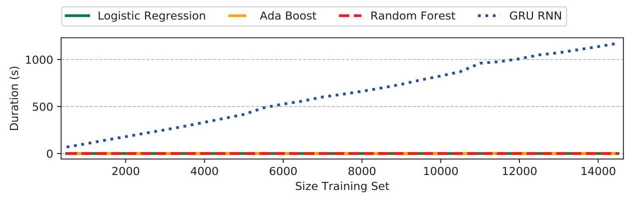


(a) 16-bit RRAM Arbiter PUF.

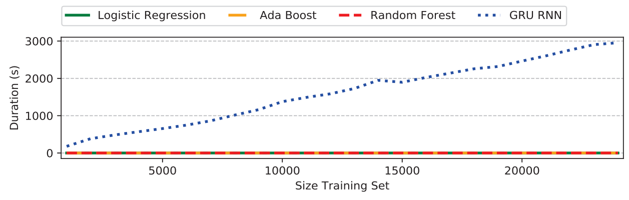


(b) 24-bit RRAM Arbiter PUF.

Fig. 5. Prediction Accuracy on RRAM Arbiter PUF.



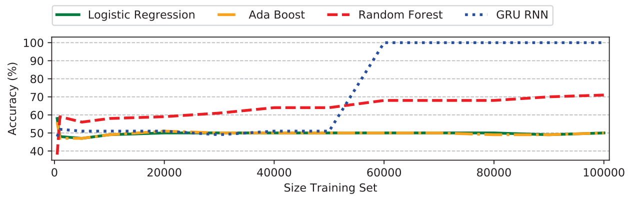
(a) 16-bit RRAM Arbiter PUF.



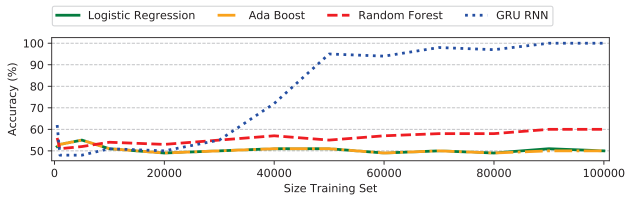
(b) 24-bit RRAM Arbiter PUF.

Fig. 6. Model Learning Duration for the RRAM Arbiter PUF.

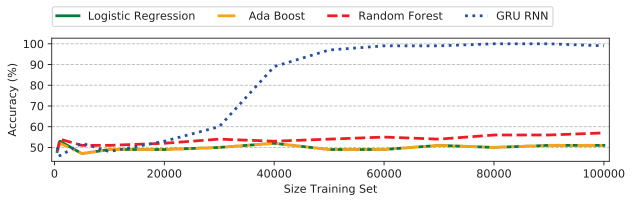
mrPUF XOR version. As described in § IV, the mrPUF has a small space of $\frac{N!}{2\times(N-2)!}$ unique CRPs, where N is the number of rows. We therefore implemented and simulated an XOR version of the mrPUF [18]. The XOR version has an increased CRP space by a power of 2. Our goal is to verify if such an improved design could be used as a strong PUF. The results of our modeling attacks are shown in Fig. 7. The PUF model can be successfully learned with the GRU RNN for all simulated PUF sizes with an accuracy of $\approx$ 100%. Random Forest only achieves a prediction accuracy of 70% for the 40 rows PUF, 60% for the 64 rows PUF and 57% for the 128 rows PUF. Logistic Regression and AdaBoost do not perform better than random guessing. In comparison to the RRAM Arbiter PUF and Memristor Arbiter PUF, a larger number of CRPs is needed to correctly learn the mrPUF model. Learning the model with GRU RNN using 100,000 CRPs would take a weak adversary approximately 9 days. However, for an advanced attacker with more computational power, the learning time is reduced. The results shown in Fig. 7 are created on a Tensorbook laptop with Intel Core i7-8750H CPU, 32 GB RAM and a NVIDIA GTX 1070 GPU, which is optimized for deep learning tasks. On the Tensorbook, learning the model of the 128 rows PUF takes only 70 s. Therefore, an attacker that can utilize the computational power of many machines or even computing clusters will be also able to break larger PUFs.



(a) 40 rows mrPUF xor version.



(b) 64 rows mrPUF xor version.



(c) 128 rows mrPUF xor version.

Fig. 7. Prediction Accuracy on mrPUF xor version.

RRAM Arbiter PUF XOR version. As shown in [23], augmenting APUFs with XOR gates does not increase the complexity of the PUF design drastically. Therefore, XOR-APUFs [21] could still be learned [6]. Our experiments show similar results for XOR versions of the mrPUF [18] and the RRAM Arbiter PUF [14]. For the latter, we tested the XOR version of both PUF sizes 16-bit and 24-bit. Our initial results on relatively small sets of CRPs are promising. We achieved a prediction accuracy of 90% for the 16-bit and of 86% for the 24-bit RRAM Arbiter PUF using GRU-RNN. For these initial results, we used a CRP set consisting of 29,000 CRPs, which is the complete set of CRPs we had available when writing this paper. The reason for the relatively small CRP set is the long simulation time of the RRAM Arbiter PUF (770 s per CRP for the 24-bit PUF).

Apart from our results, increasing the number of XOR gates that can be used in a PUF could lead to the loss of its reliability [6], [7]. Moreover, adding XOR gates to a PUF design up to the point where it is believed that the learning is infeasible due to limitations of current computational resources is also a brittle approach.

Effect of noisy responses on PUF Modeling. We tested the effect of noisy responses on the learned model for RRAM Arbiter PUF, since it shows more complex behavior than the other tested PUFs. For that, we randomly flipped 5% of the responses in the training set of a 24-bit RRAM Arbiter PUF, i.e., assuming the percentage of noisy responses is 5%, which is worse than the reliability results of 0.13% reported in [14]. The achieved prediction accuracy dropped only to 92% using GRU RNN.

VII. Related Work

Since their introduction, PUFs have been the target of various attacks ranging from physical attacks (invasive and side-channel based) [3]–[5] to software-based modeling attacks. Strong PUFs are prone to modeling attacks since they possess an exponential space of input-output (CRPs). During the last decade, several novel (e.g., Current-Mirror PUF [32] and Bistable Ring PUF [33]) or improved PUF designs (e.g., XOR-APUF [21] and Feed Forward APUF [22]) were introduced and later proven to be insecure against modeling attacks [6], [7], [25], even when the underlying mathematical model of the PUF is unknown [34]. Moreover, hybrid attacks on PUFs combining physical attacks and machine learning techniques were also proven to be effective. For example, power and timing side-channels [26] or fault injection techniques [35] can be used to reduce the complexity of modeling attacks. Ganji et al. [23] proved that the number of CRPs required for learning XOR arbiter PUF models increases only *polynomial* in process variation and number of stages. Therefore, it is not possible to build secure (XOR) arbiter PUFs relying on current IC technologies. However, such results are not conclusive regarding emerging technologies. The impact of the stochastic switching behavior of memristors in addition to process variations were thought to be high enough to impede modeling attacks, which motivated the researchers to investigate hybrid PUFs. Thus, several hybrid PUF designs combining emerging nano-technologies with classical delay-based PUFs promised great advantages (higher robustness and resilience to modeling attacks) over CMOS-only delay-based PUFs. However, only the Xbar PUF [12], [24] was shown later to be vulnerable to modeling attacks [36]. In this work, we show that recent PUFs incorporating memristors [14], [16], [18] are also susceptible to modeling attacks using state-of-the-art ML algorithms. We also show that even when XOR gates are used to increase the complexity of PUFs, they can still be learned with high prediction rates.

VIII. Conclusion and Future Work

Recent hybrid PUF designs leveraging emerging nano-electronic technologies are claimed to be resilient to modeling attacks. However, since these simulation-based designs are not open-source, it is harder for the research community to verify their security. In this work, we reproduced a set of recently proposed PUFs and verified their resilience to modeling attacks. Our results indicate that even when highly non-linear emerging technologies such as memristors are deployed to build modeling-resilient hybrid PUFs, it is still feasible to break them with state-of-the-art ML. For future work, we plan to evaluate the security guarantees of PUF designs that leverage the non-linear behavior of other nano-technologies, e.g., MRAM or PSM.

Acknowledgment. We thank Dr. Yansong Gao for sharing mr-PUF design to validate our results. This project was co-funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 CROSSING/236615297, the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS), and the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within ATHENE.

References

[1] Herder et al., "Physical unclonable functions and applications: A tutorial," *Proceedings of the IEEE*, vol. 102, no. 8, 2014.

[2] Delvaux et al., "A survey on lightweight entity authentication with strong pufs," *ACM Computing Surveys (CSUR)*, 2015.

[3] Helfmeier et al., "Cloning physically unclonable functions," in *HOST*. IEEE, 2013.

[4] Tajik et al., "Physical characterization of arbiter pufs," in *CHES*. Springer, 2014.

[5] J. Delvaux and I. Verbauwhede, "Side channel modeling attacks on 65nm arbiter pufs exploiting cmos device noise," in *HOST*. IEEE, 2013.

[6] Rührmair et al., "Puf modeling attacks on simulated and silicon data," *IEEE TIFS*, 2013.

[7] G. T. Becker, "The gap between promise and reality: On the insecurity of xor arbiter pufs," in *CHES*. Springer, 2015.

[8] Park et al., "High-density integration of carbon nanotubes via chemical self-assembly," *Nature Nanotechnology*, 2012.

[9] Williams, "How we found the missing memristor," *IEEE Spectrum*, 2008.

[10] Wolf et al., "Spintronics: a spin-based electronics vision for the future," *Science*, 2001.

[11] Chen et al., "Improvement of data retention in hfo 2/hf 1t1r rram cell under low operating current," in *IEDM*. IEEE, 2013.

[12] G. S. Rose and C. A. Meade, "Performance analysis of a memristive crossbar puf design," in *DAC*. IEEE, 2015.

[13] Vatajelu et al., "Stt-mram-based strong puf architecture," in *Annual Symposium on VLSI*. IEEE, 2015.

[14] Govindaraj et al., "Design, analysis and application of embedded resistive ram based strong arbiter puf," *IEEE TDSC*, 2018.

[15] Koeberl et al., "Memristor pufs: a new generation of memory-based physically unclonable functions," in *DATE*. EDA Consortium, 2013.

[16] Chatterjee et al., "Memristor based arbiter puf: cryptanalysis threat and its mitigation," in *IEEE International Conference on VLSI Design*, 2016.

[17] Graves et al., "Speech recognition with deep recurrent neural networks," in *ICASSP*. IEEE, 2013.

[18] Gao et al., "mrPUF: A novel memristive device based physical unclonable function," in *ACNS*. Springer, 2015.

[19] Guajardo et al., "Fpga intrinsic pufs and their use for ip protection," in *CHES*. Springer, 2007.

[20] Wang et al., "Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints," in *S&P*. IEEE, 2012.

[21] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *DAC*. IEEE, 2007.

[22] Lee et al., "A technique to build a secret key in integrated circuits for identification and authentication applications," in *Symposium on VLSI Circuits*. IEEE, 2004.

[23] Ganji et al., "Why attackers win: on the learnability of xor arbiter pufs," in *TRUST*. Springer, 2015.

[24] Uddin et al., "Techniques for improved reliability in memristive crossbar puf circuits," in *Annual Symposium on VLSI*. IEEE, 2016.

[25] Vijayakumar et al., "Machine learning resistant strong puf: Possible or a pipe dream?" in *HOST*, 2016.

[26] Rührmair et al., "Efficient power and timing side channels for physical unclonable functions," in *CHES*. Springer, 2014.

[27] T. K. Ho, "Random decision forests," in *ICDAR*, 1995.

[28] R. E. Schapire and F. Park, "A brief introduction to boosting," in *IJCAI*. ACM, 1999.

[29] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *MWSCAS*. IEEE, 2017.

[30] A. Rák and G. Cserey, "Macromodeling of the memristor in spice," *IEEE TCAD*, 2010.

[31] Kvatinsky et al., "Team: Threshold adaptive memristor model," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2013.

[32] R. Kumar and W. Burleson, "On design of a highly secure puf based on non-linear current mirrors," in *HOST*. IEEE, 2014.

[33] Chen et al., "The bistable ring puf: A new architecture for strong physical unclonable functions," in *HOST*. IEEE, 2011.

[34] Ganji et al., "Strong machine learning attack against pufs with no mathematical model," in *CHES*. Springer, 2016.

[35] J. Delvaux and I. Verbauwhede, "Fault injection modeling attacks on 65 nm arbiter and ro sum pufs via environmental changes," *IEEE TCAS1*, 2014.

[36] Uddin et al., "Robustness analysis of a memristive crossbar puf against modeling attacks," in *IEEE TNANO*, 2017.

D LO-FAT: LOW-OVERHEAD CONTROL FLOW ATTESTATION IN HARDWARE

[4] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, N. Asokan, Ahmad-Reza Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In Proceedings of the 54th ACM/IEEE Design Automation Conference (DAC'17), 2017.

LO-FAT: Low-Overhead Control Flow ATtestation in Hardware

Ghada Dessouky[1], Shaza Zeitouni[1], Thomas Nyman[2,3], Andrew Paverd[2], Lucas Davi[4], Patrick Koeberl[5], N. Asokan[2], Ahmad-Reza Sadeghi[1]

[1]Technische Universität Darmstadt, Germany
{ghada.dessouky,shaza.zeitouni,ahmad.sadeghi}@trust.tu-darmstadt.de

[2]Aalto University, Finland
thomas.nyman@aalto.fi, andrew.paverd@ieee.org, asokan@acm.org

[3]Trustonic, Finland
thomas.nyman@trustonic.com

[4]University of Duisburg-Essen, Germany
lucas.davi@uni-due.de

[5]Intel Labs, Germany
patrick.koeberl@intel.com

ABSTRACT

Attacks targeting software on embedded systems are becoming increasingly prevalent. Remote attestation is a mechanism that allows establishing trust in embedded devices. However, existing attestation schemes are either static and cannot detect control-flow attacks, or require instrumentation of software incurring high performance overheads. To overcome these limitations, we present LO-FAT, the first *practical hardware-based* approach to control-flow attestation. By leveraging existing processor hardware features and commonly-used IP blocks, our approach enables efficient control-flow attestation without requiring software instrumentation. We show that our proof-of-concept implementation based on a RISC-V SoC incurs no processor stalls and requires reasonable area overhead.

1 Introduction

Embedded systems have been facing a variety of security challenges for decades [25] which are becoming increasingly prevalent with emerging trends such as collaborative Internet of Things (IoT). A recent prominent example is *Mirai* malware[1] in October 2016, where a series of Distributed Denial-of-Service (DDoS) attacks against the DNS system disrupted a number of prominent websites.These attacks were perpetrated by IoT devices, including routers, DVRs, and web-enabled security cameras, that had been compromised by the Mirai malware.

Increasingly, attacks against embedded systems aim to exploit software vulnerabilities. In 2015, a remotely exploitable buffer overflow vulnerability was found in the *USB over IP* software used in millions of residential gateways and wireless routers supplied by prominent manufacturers[2]. In 2014, a memory corruption flaw was found in the embedded webserver software used by over 200 different models of embedded devices, affecting at least 12 million devices, many of which still remain vulnerable today[3].

Remote attestation is an important class of security mechanisms designed to detect software attacks. In principle, remote attestation allows one entity (the *verifier*) to ascertain the precise state of the software running on a remote system (the *prover*). However, most attestation schemes are *static* in that they attest the software initially loaded by the prover before it begins executing. Although useful, this still leaves the system vulnerable to *run-time* software attacks. If the adversary gains control of the stack or heap, (s)he can alter control-flow information to subvert the control flow of the target program, and mount a *code-reuse attack*. Similarly, in *non-control data* attacks [8], the adversary modifies strategic data variables to cause a permissible but unintended control flow change (e.g., executing a privileged instruction sequence). Traditionally, code-reuse attacks are mitigated using techniques such as control-flow integrity (CFI) [1]. However, CFI cannot prevent non-control data attacks, since these do not violate control-flow integrity. Neither of these types of attacks can be detected by static attestation.

To overcome these challenges *control-flow attestation* [2] was proposed very recently, enabling the prover to precisely report the control flow of application software to the verifier while giving assurance on control-flow integrity and detection of non-control data attacks. The attestation mechanism of [2] requires an isolated execution environment (e.g., ARM TrustZone, Intel SGX) to protect it against potentially compromised application software. However, implementing control-flow attestation in software has two limitations: Firstly, in order to detect control-flow events, the application software must be *instrumented* prior to deployment. Non-instrumented or incorrectly-instrumented software cannot be attested. The instrumentation rewrites all control-flow instructions (e.g., `branch`, `return`, etc.) in order to transfer control to the attestation software. Secondly, the attestation software runs on the main processor which incurs significant performance penalties because single control-flow instructions are essentially replaced with relatively many numbers of instructions in order to track and record the control-flow event (e.g., update a running hash value). As we elaborate in §7, some existing hardware approaches, such as debugging and tracing features in modern processors [14, 24] or hardware security architectures [3, 6, 9], can be used to record control flow information. However, due to the overhead they incur or the type

[1]https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html
[2]http://blog.sec-consult.com/2015/05/kcodes-netusb-how-small-taiwanese.html

[3]http://mis.fortunecook.ie/

DAC '17, June 18 - 22, 2017, Austin, TX, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4927-7/17/06...$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062276

of information they record, these approaches are not well-suited for control-flow *attestation*.

Goals and Contributions. To overcome the limitations of a software solution, we introduce a practical hardware-based Low-Overhead Control Flow ATtestation architecture, LO-FAT. Unlike software implementations, LO-FAT can handle *unmodified application software* without instrumentation, meaning that it is transparent to legacy software. By recording the control flow in hardware in parallel to the main processor, LO-FAT does not stall the application software, thus eliminating the performance overhead of attestation in software. LO-FAT leverages existing processor features and commonly-used IP blocks and can feasibly be implemented on typical embedded systems hardware platforms.

The main contributions of this paper are:

- Design of LO-FAT, a hardware-based scheme for control-flow attestation, providing the *same security guarantees* as previous software schemes, without the performance overhead or the need for software instrumentation (§4).
- An integrated optimization for eliminating redundant attestation computation (e.g., avoiding duplication when attesting loops) and reducing the burden on the verifier (§4).
- A proof-of-concept implementation of LO-FAT on the new open-source RISC-V architecture targeting the Pulpino core for single-threaded embedded system software (§5).
- A systematic evaluation of LO-FAT in terms of the required hardware area and performance benefits (§6).

2 Problem Setting and Challenges

Remote attestation provides a well-known mechanism for detecting malware on a device. However, existing conventional (binary) attestation cannot detect run-time exploitation techniques, since run-time attacks do not not modify the program binary. Such attacks aim to subvert the intended control flow of the targeted program while it is executing. An overview of different classes of such attacks is shown in Figure 1. In general, a program reserves dedicated memories for data and code. The former is marked as readable and writable *(rw)*, whereas the latter is as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten. Furthermore, any program can be abstracted through its corresponding control-flow graph (CFG) that encapsulates the valid paths a program should follow at run-time.

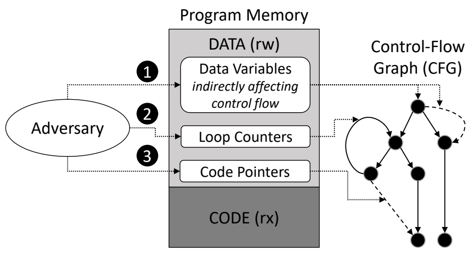


Figure 1: Overview on run-time attack classes

We can distinguish three classes of run-time attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop counter variables, and ❸ code-pointer overwrites. The most prominent run-time attacks exploit code-pointer overwrites, i.e., corruption of return addresses and function pointers. For instance, code-reuse attacks such as *Return-oriented Programming* (ROP) [23] exploit memory corruption vulnerabilities (e.g., buffer overflows) in the program and then stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the vulnerable program memory. This is exemplified by a malicious CFG edge (see dashed line for code-pointer overwrite in Figure 1). These attacks have been shown

Verifier $\mathcal{V}$ **Prover** $\mathcal{P}$

$id_S,\ i,\ CFG(S)$ $S,\ I$

$\xrightarrow{id_S,\ i,\ N}$

$P = (A, L) \leftarrow \mathsf{exec}(S(i, I))$, **where:**

$A = \mathsf{hash}([\mathsf{Src}_0, \mathsf{Dest}_0], \cdots, [\mathsf{Src}_n, \mathsf{Dest}_n])$

$L =$ **loops metadata**

$R \leftarrow \mathsf{sign}(P||N; sk)$

$\xleftarrow{P,\ R}$

$(\bot, \top) \leftarrow \mathsf{versig}(R; pk)$

$(\bot, \top) \leftarrow \mathsf{ver}(P, CFG(S)_{|i})$

Figure 2: Attestation protocol of LO-FAT

to be a realistic threat on many processor architectures, such as Intel x86 [23], ARM [17] and embedded systems building on Atmel AVR [12]. Although countermeasures against this class of attacks exist, e.g., control-flow Integrity (CFI) [1] and code-pointer integrity (CPI) [16], they do not prevent attacks ❶ and ❷. The so-called *non-control data attacks* [8] do not compromise the control flow of a program, but cause unexpected malicious control-flow paths by corrupting data variables. In ❶, the attacker compromises data variables that are used for security decisions during program execution, e.g., corrupting an authentication variable to execute a privileged but existing path. Attack class ❷ is even more subtle as it only affects the number of times a program loop is executed. This can have severe consequences in the context of embedded system software, e.g., a syringe pump dispenses more liquid than requested (see [2]).

Control-flow attestation can cover these cases by assuring the verifier of the precise run-time control flow of the program on the embedded device. In [2], the first control-flow attestation scheme was proposed and implemented. However, it suffers from practical limitations, such as high performance overhead and the need for tedious software instrumentation.

Our work tackles the challenge of detecting attack classes ❶-❸, while addressing the limitations of recently proposed software-based control-flow attestation [2] by presenting LO-FAT, an efficient hardware-only solution.

3 System Model

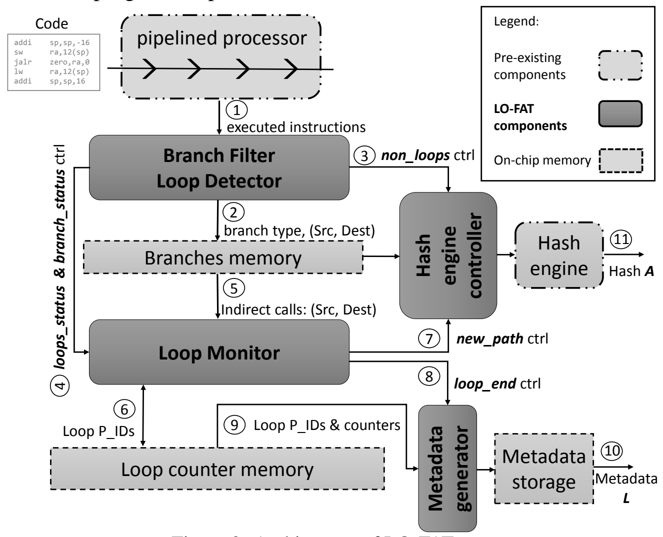
Figure 2 depicts the attestation protocol of LO-FAT: the verifier $\mathcal{V}$ aims to attest the run-time control-flow (execution path) of the Program S on a remote embedded system – the prover $\mathcal{P}$. We assume that both $\mathcal{V}$ and $\mathcal{P}$ have access to the program S in binary form and that conventional static (binary) attestation assures $\mathcal{P}$ is executing the correct and unmodified program S.

First, $\mathcal{V}$ performs a one-time offline pre-processing step to generate the CFG of S (including expected loop execution information) by means of static or dynamic analysis. Next, $\mathcal{V}$ initiates the protocol by sending $\mathcal{P}$ the program input i for the program ID id_S, and the nonce N to ensure freshness of the attestation response. $\mathcal{P}$ executes S with verifier input i and a set of malicious adversary inputs I. In fact, the untrusted inputs received may corrupt the control-flow by means of the attack techniques described in §2. While S executes, LO-FAT captures the control-flow transitions and generates a cumulative authenticator A of the control-flow path taking source and destination address $(Src, Dest)$ of each branch as input. Naively storing and transmitting every single executed instruction to $\mathcal{V}$ would incur impractical memory, power and communication overheads, especially for resource-constrained embedded devices. Hence, LO-FAT follows the idea outlined in [2] and computes a cumulative cryptographic hash of the executed path. In addition, it also produces auxiliary metadata L to track program loop paths and their number of iterations (including recursive functions) thereby covering attacks of class ❷ in Figure 1. Together A and L form a unique program path P. Lastly, upon program exit, $\mathcal{P}$ generates the *attestation report* $R = sign(P||N; sk)$, under the signing key sk, which is stored by $\mathcal{P}$ in hardware-protected secure memory, e.g., a register that is accessible only to LO-FAT. Upon receiving R, $\mathcal{V}$ verifies the signature using the verification key pk. Next, $\mathcal{V}$ checks whether the reported path P resembles a valid path in CFG under input i. If true, $\mathcal{V}$ is assured of $\mathcal{P}$'s execution.

Adversary Model and Assumptions. We assume a strong adversary that has full control over the *data memory* of $\mathcal{P}$ and can utilize standard memory corruption vulnerabilities to modify arbitrary writable memory locations. However, the adversary cannot modify program code at run-time (marked as *rx*) and cannot modify memory used by LO-FAT itself (due to hardware protection). Note that similar to all attestation schemes we consider software-only attacks and hence physical attacks on $\mathcal{P}$'s device are out of scope in this work. Also note that our scheme can detect attacks that affect the program's control-flow, but not pure data-driven attacks (that do not affect any control-flow) such as *data-oriented programming* attacks, which remain an open research problem [13].

4 LO-FAT Design

Figure 3 illustrates our architecture for LO-FAT and how it interfaces with the processor pipeline. The proposed scheme exploits branch tracking functionality inherent in any processor pipeline and re-usable IP cores such as the hash engine. We extend these with additional logic to achieve efficient tracing of control-flow information. The main LO-FAT components are the branch filter and the loop monitor. The former extracts branch instructions from the processor as it executes the attested code segment while the latter monitors program loops.



```
addi    sp,sp,-16
sw      ra,12(sp)
jalr    zero,ra,0
lw      ra,12(sp)
addi    sp,sp,16
```

Figure 3: Architecture of LO-FAT.

Branch Filter. Upon code execution, the branch filter, which is tightly coupled to the processor, extracts the current program counter and instruction executed per clock cycle. Then it filters in every `branch`, `jump` and `return` instruction since these are the relevant instructions for control-flow attestation. The branch filter outputs a concise representation of every executed branch instruction with its source and destination address pair $(Src, Dest)$ into a dedicated branches memory and detects whether the intercepted branch is within a program loop. If not, the branch filter enables hashing of $(Src, Dest)$. Branches inside a program loop require special treatment in LO-FAT, because (i) loop counter manipulation may compromise the program's control-flow in a malicious way (§2), and (ii) naively hashing each loop iteration and path leads to a combinatorial explosion of valid hash values [2]. As such, we design LO-FAT to compress control-flow information associated with loops efficiently. As mentioned earlier in §3, we report each loop path and its number of iterations as auxiliary metadata L. However, doing so in hardware is challenging, i.e., in contrast to the most related work C-FLAT, since we do not use code instrumentation to preserve legacy compliance. Hence, the branch filter must detect and identify loop entry and exit points and their depth at run-time without instrumentation aid. We describe in §5.1 how we tackled this challenge.

Loop Monitor. When a loop is encountered, the branch filter forwards the loop entry and exit to the loop monitor. The loop monitor identifies and tracks program loops (including nested loops). When a branch inside a program loop is encountered, the branch filter forwards this information to the loop monitor which in turn encodes each path inside the loop uniquely. Simultaneously, $(Src, Dest)$ of each `branch` remains stored in the branches memory.

Another major challenge concerning loops is the hash computation and attestation overhead incurred by hashing each loop iteration. In LO-FAT, we significantly reduce the hash computation cost by only hashing each loop path once and keeping an iteration counter for each unique loop path. To achieve this, LO-FAT generates a unique path encoding for each loop path and associates an on-chip loop counter with it. The loop monitor indicates newly observed loop paths to the hash engine controller in order to hash its corresponding $(Src, Dest)$ from the branches memory. On the other hand, once the same loop path executes, LO-FAT only needs to increment the counter, i.e., not requiring further hash operations.

Upon loop exit, the loop monitor requests the metadata generator to assemble the loop auxiliary metadata based on the loops memory which contains the unique loop path encodings, their number of iterations, and indirect branch targets. This information is stored on-chip and is appended to the final hash value A computed at the end of the attested execution. Finally, a digital signature R is computed over the hash value A, metadata L and nonce N and sent to $\mathcal{V}$ for attestation (as per our protocol outlined in §3).

5 Implementation

5.1 Loop Handling

Detecting loops. As shown in Figure 3, the branch filter unit traces the instruction (and its address) executed per clock cycle and filters in ① every `branch`, `jump` and `return` instruction. It outputs a concise representation of every executed branch instruction with its $(Src, Dest)$-pair into a dedicated branch buffer (②). To compress the control-flow trace for loops, the branch filter has to detect loops. If the intercepted branch is not in a loop, the branch filter sends the control signal *non_loops_ctrl* to the existing hash engine controller to compute a hash over $(Src, Dest)$ in ③. Otherwise, the branch filter forwards the loop status (entry and exit) to the loop monitor and its depth (in case of nested loops) via the *loops_status_ctrl* signals (④).

To enable efficient run-time loop detection, we utilize a property of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. LO-FAT uses a simple heuristic to differentiate between backward branches that constitute loops, and branches for subroutine calls where the call target resides earlier in memory. Since subroutine calls use instructions that update the *link-register*, we consider the target of each *non-linking* backwards branch as a *loop entry node*. The basic block proceeding the branch instruction is considered a *loop exit node*. We base our heuristic on our observations of the RISC-V compiler assembly and the calling convention described in the instruction manual: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site are still compiled as a *linking* branch or are optimized by traditional inlining using the RISC-V compiler.

The addresses of the entry and exit nodes of each loop are stored in registers by the loop detector and used to detect and track loop iterations and loop depth at run-time when executing nested loops. The number of loop iterations is determined by recording the number of times the loop entry node is entered within the loop. Loop termination is detected by tracking if execution proceeds to or past the currently active loop exit node, either as the result of sequential execution (e.g. in the case of a conditional branch) or a non-linking branch (e.g. break). Loop execution status is forwarded using the *loops_status_ctrl* signals to the loop monitor, as shown in Figure 3.

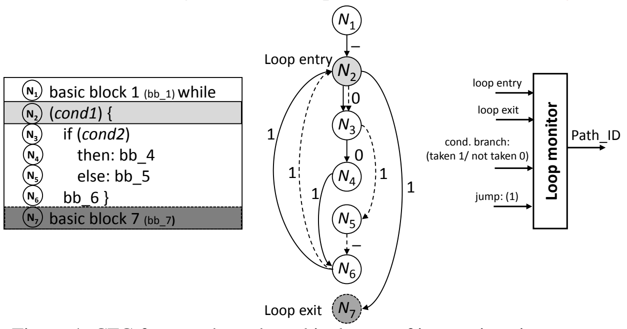


Figure 4: CFG for pseudo-code and its layout of instructions in memory.

Tracking loops. As shown in Figure 3, the loop monitor receives *branch_status_ctrl* signals from the branch filter to describe the type of intercepted branch instruction and its $(Src, Dest)$ ($\textcircled{5}$). This branch tracking mechanism allows the loop path encoder to uniquely encode paths as they occur. Simultaneously, $(Src, Dest)$ of each `branch` along the executing loop path remain stored in the branches memory.

Figure 4 shows a sample pseudo-code and its CFG according to how the instructions would be laid out in code memory to illustrate how the loop monitor encodes the loop paths. The example code shows a *while-loop* with an *if-else* statement inside. Each basic block in the pseudo-code is represented by a node in the CFG and numbered accordingly, with loop entry and exit nodes also indicated. Within this simple loop, there are only 2 valid paths: bold path $N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6 \rightarrow N_2$ and dashed path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6 \rightarrow N_2$.

For every `conditional branch`, the processor evaluates the condition and either jumps to the computed target address (branch is taken), or continues sequentially to the next instruction address in memory (branch is not taken). Processors commonly track this branching behavior in the pipeline and may encode a taken/not-taken branch with '1'/'0'. This branch information is extracted from the processor by the branch filter and used by the loop monitor to uniquely identify and encode paths within each loop with a unique path_ID, as shown in Figure 4. In Figure 4, the dashed path $N_2 \rightarrow N_3 \rightarrow N_5 \rightarrow N_6 \rightarrow N_2$ is encoded as '011' and bold path $N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_6 \rightarrow N_2$ as '0011'. Other path encodings are considered invalid and detected by the $\mathcal{V}$.

Once a loop path is completed, this unique path_ID is used to index *loop counter* memory, in which the number of iterations for each corresponding path is saved ($\textcircled{6}$) in Figure 3. A counter value of zero indicates the first time a particular path is executed. This is forwarded by the loop monitor into the hash engine controller using *new_path_ctrl* signals ($\textcircled{7}$) to enable hashing of corresponding $(Src, Dest)$ pairs. Otherwise, the counter is simply incremented.

To ensure constant-time, single-cycle memory access latency, we implement *loop counter memory* as on-chip memory indexed by the unique loop path encodings. However, this consumes a dedicated sparsely-utilized memory which is often a constrained resource on low-end embedded devices. In light of this, LO-FAT allows configuring the granularity of the control-flow tracking according to the availability of memory resources.

Once a loop exits, this is identified by the loop monitor and indicated in the *loop_end_ctrl* signals sent to the metadata generator ($\textcircled{8}$). The metadata generator assembles the loop auxiliary metadata from the loops memory - this consists of the unique loop path encodings in order of first occurrence, the number of iterations of each path, and the indirect branch targets encountered in this loop ($\textcircled{9}$). This fine-grained auxiliary information on loop execution is stored on-chip ($\textcircled{10}$) and is appended to the final hash value computed at the end of the attested execution ($\textcircled{11}$). Finally, a digital signature is computed over the hash value, metadata and nonce N, and sent to $\mathcal{V}$ for attestation. Handling indirect branches in loops is yet another implementation challenge we discuss next.

5.2 Handling Indirect Branches in Loops

Indirect branches can involve any arbitrary number of targets which can never be exhaustively identified using static analysis. To uniquely identify loop paths with indirect branches (calls and returns), we would need to include the 32-bit target addresses into the path encodings, which would require infeasibly high memory requirements for loop path-indexed memory. Instead, we re-encode the addresses using a smaller number of n bits, allowing a maximum number of 2^n-1 possible targets for each loop. Target addresses are encoded at run-time and stored in a register file, which is implemented as 2 interleaved CAMs to ensure low-latency constant-time access. When a target address is encountered that exceeds the configured limit, we report this in the encoding to the $\mathcal{V}$ by an all-zero code. LO-FAT is designed such that the maximum number of branches per loop path and the maximum number of possible target addresses (of indirect branches) to track is configurable in a trade-off between granularity and availability of on-chip memory. Tracking ℓ branches per path in a loop requires $8 \times 2^\ell$ bits memory. In our implementation, we configure $n = 4$ to track up to 16 possible indirect branch targets for a given loop and $\ell = 16$ such that LO-FAT can handle a maximum of 16 branches per loop path (every additional indirect branch tracked reduces the maximum number of possible conditional branches by n) and depth of up to 3 nested loops, which requires a dedicated 1.5 Mbits memory that is synthesized as block RAM (BRAM) when prototyping on FPGA. Once a loop exists, its memory is re-used for other subsequent loop executions.

Loop metadata. The measurement in A is a single hash computation of $(Src, Dest)$ pairs of executed loop paths. To enable $\mathcal{V}$ to reconstruct the final hash value, metadata L of the loops serves as helper data and provides $\mathcal{V}$ with fine-grained insight into the execution of the loops. L contains the encodings of executed paths in each loop, the order of first occurrence of each executed path, and number of iterations per loop path and indirect branch targets.

5.3 Hash Engine

A single hash measurement A is computed on the full execution path, along with auxiliary loop metadata L. We employ a SHA-3 512-bit open-source engine[4] operating at a maximum clock frequency of 150 MHz. It consists of a permutation module which operates on a message block size of 576-bit. User input is absorbed by the core first into a padding module to assemble the 576-bit block size. Once this padding is full, the permutation module begins computation on

[4]http://opencores.org/project,sha3

input. In LO-FAT, the engine can absorb a 64-bit input $(Src, Dest)$-pair every clock cycle into the padding module for 9 clock cycles, after which the 576-bit buffer becomes full and notifies the permutation module to begin its computation. Once full, the padding buffer cannot absorb further input for 3 clock cycles after which it resumes normally. Therefore, a small cache buffer is configured at the hash engine input to prevent dropping of $(Src, Dest)$-pairs if they arrive during these cycles where the padding buffer is full. Using this hash engine, an unlimited message size can be hashed while indicating the end of streaming $(Src, Dest)$-pairs when the execution of attested software is completed.

6 Evaluation

We present a proof-of-concept implementation of LO-FAT on Pulpino [18], the first open-source RISC-V-based microcontroller SoC [19]. It is based on a single 32-bit 4-stage minimal RISC-V core targeting low-end embedded systems. We augment the RISC-V processor pipeline to interface with the LO-FAT branch filter to extract control-flow signals required for execution flow tracing. LO-FAT can be easily integrated into any low-end embedded processor as it does not require modifications to the ISA.

6.1 Functionality and Performance

We integrated LO-FAT with Pulpino and performed cycle-accurate functional simulation of their RTL Verilog source code on ModelSim while Pulpino executed extracted code segments from real embedded applications, such as Open Syringe Pump[5], an open-source open-hardware syringe pump design. Simulation results confirmed the functionality of LO-FAT in correctly capturing and compressing the control flow (branches, loops, and nested loops) of an uninstrumented application. Since LO-FAT extracts and filters control-flow events in parallel with the processor, it does not incur any performance overhead for the attested software, as opposed to C-FLAT which incurs attestation overhead that is linearly dependent on the number of control-flow events. LO-FAT internally incurs latency of 2 clock cycles for branch instructions and loop status tracking and 5 clock cycles at loop exit for completing path_ID generation and *loop counter* memory access and update. However, LO-FAT simultaneously continues to absorb and process any incoming $(Src, Dest)$-pairs to prevent the processor from stalling or dropping trace information. Synthesis results using Xilinx Vivado indicate LO-FAT can operate at maximum clock frequency of 80 MHz on a Virtex-7 XC7Z020 FPGA device on a Zedboard. The LO-FAT units are engineered such that they operate on par with Pulpino's clock frequency, while also allowing single-cycle constant-time memory accesses for indirect branches and loops management. Eliminating the CAM access results in a much higher clock frequency if desired.

The length of the auxiliary metadata (L) that must be sent to $\mathcal{V}$ depends on the number of loops executed, the number of different paths per loop, and the number of indirect branch targets encountered in the attested code.

6.2 Area

On a Virtex-7 XC7Z020, LO-FAT consumes 4% of the available registers and 6% of available LUTs, which amounts to an average of 20%additional logic overhead to the Pulpino SoC. 49 36Kbit Block RAM (BRAMs) are utilized, most of which are dedicated for the sparse loop path-indexed memories to ensure constant-time single-cycle access. Therefore, its width depends on the configured maximum number of indirect branches allowed in each loop path and number of bits required to encode them, as discussed in §5.2. In our implementation, the loop monitor is configured to tackle up to 4 indirect branches and requires 10 bits to encode them in $Path\_ID$, resulting in 16 BRAMs per loop. Since we allow up to 3 levels of nested loops, we require 48 BRAMs. Configuring these parameters to lower numbers or leveraging CAMs instead reduces the memory requirements significantly at the expense of coarser granularity or additional logic overhead respectively.

[5]https://hackaday.io/project/1838-open-syringe-pump

6.3 Security

The primary security requirement of LO-FAT is to provide an *accurate*, *complete*, *authentic*, and *fresh* attestation of $\mathcal{P}$'s control flow. This requires an integrity-protected mechanism for recording control-flow information and unforgeably communicating this to $\mathcal{V}$.

Control-Flow Recording. One of the main contributions of LO-FAT is using low-overhead hardware extensions to record control-flow information preventing it from being modified or subverted by malicious software. The on-chip memory employed by LO-FAT for storing the $(Src, Dest)$ addresses prior to their hashing is also assumed to be protected from adversarial access. The hardware extensions are guaranteed to receive every control-flow event from the processor, thus ensuring that the complete control flow is recorded. All $(Src, Dest)$ addresses are cryptographically hashed resulting in the authenticator A. The auxiliary metadata L records (1) the unique paths within each loop; (2) the number of repetitions of each path; and (3) all indirect branches encountered within loops.

Attestation Protocol. LO-FAT makes use of the widely-used secure challenge-response attestation protocol. As explained in §3, $\mathcal{P}$ sends the recorded program path P along with a digital signature over P and a nonce supplied by $\mathcal{V}$. If $\mathcal{P}$'s signing key has not been compromised, this signature guarantees the authenticity of the attestation, and the inclusion of the challenge nonce ensures freshness. Our assumed software adversary cannot compromise the signing key because it is stored in hardware-protected secure memory. Any tampering with the attestation messages can be detected by $\mathcal{V}$.

Given that the control flow recording and the signing key is protected from software attacks, the resulting attestation report provided by LO-FAT is accurate, complete, authentic, and fresh. Since $\mathcal{P}$'s code is immutable and is statically attested at boot time, $\mathcal{V}$ has complete information about $\mathcal{P}$'s execution. As described in §3, $\mathcal{V}$ also has access to the CFG of the attested software, which it can use to identify permissible control flows and detect control-flow attacks or non-control data attacks.

7 Related Work

Remote Attestation. Most prior work focuses on *static* remote attestation [7, 11, 21], which is orthogonal to run-time attestation – the focus of this paper. Software-based attestation [22] can, under strict assumptions, enable static attestation of legacy devices without hardware-based trust anchors. Property-based attestation [20] can attest behavioral characteristics of a program, with the assistance of a trusted third-party. However, none of these can attest control-flow at machine code instruction level.

Prior work on run-time attestation focuses on specific aspects of a program's execution. ReDAS [15] attests program data invariants, such as the integrity of a function's base pointer, at each system call. Trusted virtual containers [4] attest the run-time launch order of application modules – a form of coarse-grained control-flow attestation that does not include internal control flows within modules. DynIMA [10] uses dynamic taint analysis and tracing to attest run-time properties that may be symptomatic of run-time attacks. However, it does not cover non-control data attacks and incurs high performance overhead due to dynamic taint analysis.

C-FLAT [2] is a fine-grained control-flow attestation scheme. LO-FAT also leverages the idea of attesting the control flow of an application by computing a cumulative hash of executed branches but with several fundamental differences. C-FLAT requires *instrumentation* of all control-flow instructions thereby violating legacy compliance. In contrast, LO-FAT does not require any binary rewriting. C-FLAT requires complete coverage in the offline binary analysis, as un-instrumented control-flow instructions could be exploited to mount undetectable attacks. This is not possible in LO-FAT as every executed branch is monitored by design. Finally, C-FLAT incurs significant performance overhead, whereas LO-FAT incurs no performance overhead due to its efficient hardware support for control-flow attestation.

Tracing and Debug Mechanisms. Intel processors provide the *Last Branch Record* (LBR) and *Branch Trace Store* (BTS) mechanisms, which can be used to trace control-flow events [24]. However, the overhead incurred by these debugging mechanisms makes them unsuitable for control-flow attestation. Recently, Intel processors introduced *Intel Processor Trace* (IPT) [14], a low-overhead execution tracing feature that collects more tracing information than BTS (including execution mode and timing information). However, IPT cannot be directly used for control-flow attestation as it only reports control-flow events that cannot be inferred from static analysis. ARM's CoreSight[6] debug and trace architecture provides a mechanism to access trace information from different hardware trace components. However, high-throughput tracing on ARM typically requires the use of proprietary hardware.

Hardware-Assisted Security. Recent work [5, 26] developed a generic architecture for enforcing a diverse range of SoC security policies. Each IP block has an individually-customized security wrapper that sends security-relevant events and information to a central security controller to enforce individual security policies for each IP. However, this incurs high memory and logic complexity overhead as the number of IPs increases. It has further been proposed [3, 6] that this could be made more practical by re-purposing design-for-debug features found on many SoCs – a promising approach which could complement LO-FAT in future.

Sofia [9] is a recent hardware-assisted architecture for enforcing control-flow integrity (CFI). It encrypts instructions with CFI-dependent data, such that they can only be decrypted at run-time as part of a valid control-flow path, and it ensures instruction integrity by checking MACs on groups of instructions at run-time. However, unlike LO-FAT, this requires software instrumentation and places decryption in the critical execution path, thus incurring total execution time overheads of up to 110%.

8 Conclusion

Due to the increasing prevalence of interconnected embedded systems, software running on these devices have become a prime target for remote attacks. We presented in this paper the first hardware-based control-flow attestation scheme that allows precise detection of remote memory corruption attacks in embedded system software. Our architecture, LO-FAT, monitors, measures and reports the program's behavior by interfacing with the processor to intercept control-flow events. LO-FAT does not require any code instrumentation (compliant to legacy software), compiler toolchain or instruction set extension. Our proof-of-concept implementation on the open-source RISC-V core is highly efficient with no performance impact on the attested software at the expense of minimal logic overhead and on-chip memory.

Acknowledgments. This work was supported by the German Science Foundation CRC 1119 CROSSING project, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant number 643964 (SUPERCLOUD), Tekes — the Finnish Funding Agency for Innovation (CloSer project), and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, pages 4:1–4:40, 2009.

[2] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *ACM CCS*, 2016.

[3] J. Backer, D. Hely, and R. Karri. On Enhancing the Debug Architecture of a System-on-Chip (SoC) to Detect Software Attacks. In *IEEE DFT*, 2015.

[4] K. A. Bailey and S. W. Smith. Trusted Virtual Containers on Demand. In *ACM-CCS-STC*, 2010.

[5] A. Basak, S. Bhunia, and S. Ray. A Flexible Architecture for Systematic Implementation of SoC Security Policies. In *ACM/IEEE DAC*, 2015.

[6] A. Basak, S. Bhunia, and S. Ray. Exploiting Design-for-Debug for Flexible SoC Security Architecture. In *ACM/IEEE DAC*, 2016.

[7] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl. TyTAN: Tiny Trust Anchor for Tiny Devices. In *ACM/IEEE DAC*, 2015.

[8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security*, 2005.

[9] R. d. Clercq, R. D. Keulenaer, B. Coppens, B. Yang, P. Maene, K. d. Bosschere, B. Preneel, B. d. Sutter, and I. Verbauwhede. SOFIA: Software and Control Flow Integrity Architecture. In *ACM/IEEE DATE*, 2016.

[10] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks. In *ACM CCS-STC*, 2009.

[11] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *ISOC NDSS*, 2012.

[12] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-architecture Devices. In *ACM CCS*, 2008.

[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-Oriented Programming: On The Effectiveness of Non-Control Data Attacks. In *IEEE S&P*, 2016.

[14] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C, Chapter 36 Intel Processor Trace. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2016.

[15] C. Kil, E. Sezer, A. Azab, P. Ning, and X. Zhang. Remote attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *IEEE/IFIP DSN*, 2009.

[16] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *USENIX OSDI*, 2014.

[17] L. Le. ARM Exploitation ROPMap. BlackHat USA, 2011.

[18] Pulpino. An Open-Source Microcontroller System based on RISC-V. https://github.com/pulp-platform/pulpino.

[19] RISC-V. The Free and Open RISC Instruction Set Architecture. https://riscv.org/specifications, 2016.

[20] A.-R. Sadeghi and C. Stüble. Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms. In *NSPW*, 2004.

[21] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security*, 2004.

[22] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based Attestation for Embedded Devices. In *IEEE S&P*, 2004.

[23] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *ACM CCS*, 2007.

[24] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting Hardware Advances for Software Testing and Debugging (NIER Track). In *ACM/IEEE ICSE*, 2011.

[25] J. Viega and H. Thompson. The State of Embedded-Device Security (Spoiler Alert: It's Bad). *IEEE S&P*, 10(5):68–70, 2012.

[26] X. Wang, Y. Zheng, A. Basak, and S. Bhunia. IIPS: Infrastructure IP for Secure SoC Design. *IEEE Transactions on Computers*, Aug 2015.

[6]https://www.arm.com/products/system-ip/coresight-debug-trace

E

ATRIUM: RUNTIME ATTESTATION RESILIENT UNDER MEMORY ATTACKS

[5] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, Ahmad-Reza Sadeghi. "ATRIUM: Runtime Attestation Resilient Under Memory Attacks". In Proceedings of the 36th ACM/IEEE International Conference on Computer Aided Design (ICCAD'17), 2017.

ATRIUM: Runtime Attestation Resilient Under Memory Attacks

Shaza Zeitouni
TU Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

Ghada Dessouky
TU Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Orlando Arias
University of Central Florida, USA
oarias@knights.ucf.edu

Dean Sullivan
University of Florida, USA
deanms@ufl.edu

Ahmad Ibrahim
TU Darmstadt, Germany
ahmad.ibrahim@trust.tu-darmstadt.de

Yier Jin
University of Florida, USA
yier.jin@ece.ufl.edu

Ahmad-Reza Sadeghi
TU Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Remote attestation is an important security service that allows a trusted party (verifier) to verify the integrity of a software running on a remote and potentially compromised device (prover). The security of existing remote attestation schemes relies on the assumption that attacks are *software-only* and that the prover's code cannot be modified at runtime. However, in practice, these schemes can be bypassed in a stronger and more realistic adversary model that is hereby capable of controlling and modifying code memory to attest benign code but execute malicious code instead – leaving the underlying system vulnerable to Time of Check Time of Use (TOCTOU) attacks.**

In this work, we first demonstrate TOCTOU attacks on recently proposed attestation schemes by exploiting physical access to prover's memory. Then we present the design and proof-of-concept implementation of ATRIUM, a *runtime* remote attestation system that securely attests both the code's binary and its execution behavior under memory attacks. ATRIUM provides resilience against both software- and hardware-based TOCTOU attacks, while incurring minimal area and performance overhead.

***Index Terms*—Attestation, runtime, memory attacks**

I. Introduction

Recent high-profile attacks on embedded systems, such as Mirai and Stuxnet, have become crucially alarming and of increased significance as systems are becoming more interconnected and collaborative. *Remote attestation* plays an important role as a security service for detecting malware on a remote device. It is implemented as a challenge-response protocol that allows a trusted *verifier* to obtain an authentic report about the (software) state of a potentially untrusted remote device called *prover*. Conventional attestation schemes are static in nature, i.e., the prover sends an authenticated report to the verifier by issuing a digital signature or cryptographic MAC (Message Authentication Code) over the verifier's challenge and the *measurement* (typically hash) of the binary code to be attested [22]. However, static attestation only ensures the integrity of binaries but *not* of their execution. In particular, it cannot detect the prevalent state-of-the-art runtime attacks that do not modify the program binary but subvert the intended control flow of the targeted application program during its execution. Current runtime attacks take advantage of code-reuse techniques, such as return-oriented programming that dynamically generate malicious code by chaining together code snippets (called gadgets) of benign code *without* requiring to inject any malicious code/instructions [24]. Consequently, the hash value computed over the binaries remain unchanged and the attestation protocol succeeds, although the prover has been compromised. These sophisticated exploitation techniques have been shown effective on many processor architectures, such as Intel x86 [23], SPARC [4], ARM [16], and Atmel AVR [10]. In fact, large-scale investigations of embedded systems security have shown various vulnerabilities, including memory corruption (such as buffer overflow) that can be exploited for runtime attacks.

Hence, effective attestation should enable reporting the prover's dynamic behavior – more concretely, its current execution details – to the verifier. To attest the dynamic program behavior researchers have proposed enhancements and/or extensions to static binary attestation (e.g., [11], [3]). The most recent, C-FLAT [3], reports the prover's dynamic state (execution paths) and provides fine-grained control-flow measurements to the verifier. Note that, unlike control-flow integrity (CFI) enforcement, control-flow attestation provides detailed information about the executed path that might be of crucial interest to a remote verifier. This information helps in detecting data-oriented non-control attacks [5] that can bypass CFI by corrupting data variables to execute a valid but unintended control-flow path, for instance, redirecting the control flow to a high-privileged recovery routine (see also [13]). However, C-FLAT requires program code instrumentation and incurs high performance overhead, particularly on the prover.

On the other hand, all existing attestation schemes (including C-FLAT) rule out physical attacks in their adversary model. This assumption is not always realistic, since the adversary may at some point have physical access to the prover. In this case, it is possible to execute (extraordinarily effective and cheap) non-invasive attacks on the program code memory through *physical access*. In particular, the adversary physically controls and modifies the memory such that benign code is attested but malicious code is executed instead.

Goals and Contributions. In this paper, we first demonstrate that – using external interfacing with prover's program code memory bank – an adversary can bypass all existing attestation schemes and deliver sound attestation reports, without even having to extract the prover's secret keys (cf. § III). To overcome the limitations of current attestation schemes, we introduce a holistic approach to attestation ATRIUM, a *resilient runtime attestation* scheme that is capable of detecting both physical memory attacks and software attacks including runtime attacks by attesting the executed instructions and their control flow at runtime. Our main contributions are listed as follows.

- We demonstrate memory bank attacks on state-of-the-art attestation schemes for embedded devices such as SMART [9] and C-FLAT [3]. We exploit physical access to code memory to bypass attestation and deliver sound attestation reports without having to extract the prover's secret keys.
- We present ATRIUM– an attestation scheme which: (1) detects memory bank attacks by attesting instructions as they are fetched from (off-chip) memory for execution; (2) prevents software attacks on the attestation process itself by separating the attestation engine from the processor (i.e., no instructions are sent to the processor to perform attestation). Instead, attestation is performed by a separate hardware engine in parallel. (3) detects runtime attacks by tracking and reporting both executed instructions and control-flow events during execution.
- We present a proof-of-concept implementation and performance analysis which demonstrate the effectiveness and feasibility of ATRIUM, and its applicability to low-end embedded devices.

II. Background

Control-Flow Graph (CFG). The execution flow of a program can be abstracted into a control-flow graph (CFG) by leveraging the aid of static and dynamic code analysis. The nodes in CFG represents basic blocks of a program, while edges represent control-flow transitions from one block to another by means of a branch instruction. A *valid* path in CFG is composed of several nodes connected by edges.

Runtime Attacks. An outline of the different classes of runtime attacks is illustrated in Figure 1. The system dedicates separate memories for data and code. The former is marked as readable and writable *(rw)*, while the latter is marked as readable and executable *(rx)*. This ensures that code cannot be executed from data memory, and code memory cannot be overwritten *by means of software*. Along this CFG, we can outline three major classes of runtime attacks: ❶ non-control-data attacks that indirectly affect the control flow of a program, ❷ corruption of loop variables, and ❸ code-pointer overwrite attacks. By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses and function pointers) as in ❸ an attacker can redirect the control flow of a program such that execution has a malicious and unauthorized effect. In attacks based on *code-injection*,

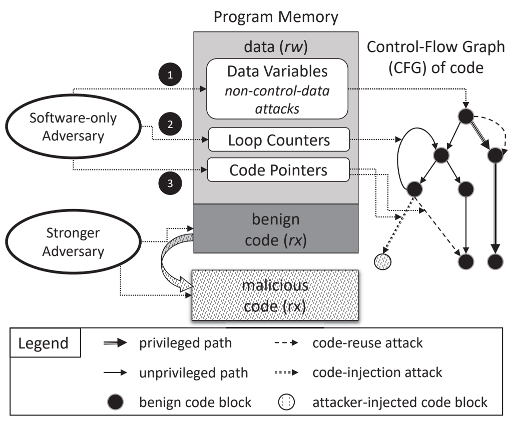


Figure 1: Different attack classes

the attacker places a malicious executable payload in program memory and redirects control flow to execute it. Alternatively, state-of-the-art runtime attacks leverage *code-reuse* techniques, such as *Return-oriented Programming* (ROP) [23]. These attacks exploit a memory corruption vulnerabilities (e.g., buffer overflows) in the program and stitch together a malicious sequence of machine code instructions from benign *gadgets* of code already residing in the memory of the vulnerable program. *Non-control-data attacks* [5] do not compromise the control flow of a program, but cause unexpected malicious control flow by corrupting critical data variables such as an authentication variable. This results in executing a privileged (unintended) but permissible control-flow path that exists in the CFG. Attack ❷ affects the number of times a program loop executes by corrupting a loop variable such as a counter. This can have severe consequences depending on the context, e.g., a syringe pump dispenses more liquid than requested (see [3]). Code injection attacks can be prevented by either marking memory as writable or executable. This mechanism is known as *Data Execution Prevention* (DEP) [12]. Countermeasures against code reuse attacks include: *Control-Flow Integrity* (CFI) [2], fine-grained code randomization [19], and Code-Pointer Integrity (CPI) [18].

Besides software-based runtime attacks, a stronger adversary as shown in Figure 1, can modify program code in memory through *physical access* without mounting sophisticated invasive physical attacks, but by simply replacing the benign code memory with malicious code memory at runtime. We elaborate on these memory bank attacks next in § III and propose an attestation scheme that can mitigate them in § V.

III. TOCTOU Attacks on Attestation Schemes

Next we describe memory bank attacks that we aim to mitigate in this work, and we show how they bypass recently proposed attestation schemes: SMART [9] C-FLAT [3] and LO-FAT [7]. These attacks assume a stronger adversary that can physically manipulate the code memory without the need for sophisticated invasive physical attacks and can consequently bypass attestation schemes that strictly consider software-only adversary. The attack is illustrated in Figure 2: At $\mathcal{P}rv$'s side the attestation scheme (i.e., the attestation code and secret key) is stored on-chip while the benign code resides in an external memory. The adversary can interleave instruction fetches to malicious code in-between those fetches needed to attest the benign code of the original program. This can be done by replacing the original memory interface with an interface to a memory controller. This allows the adversary to direct instruction fetches to either benign code when attestation is running, or malicious code otherwise. The same interleaving attack can be achieved by inserting malicious instructions in-between hooks to the attestation. As long as the malicious instructions do not interfere with attesting benign code, e.g., intended control flow, the attestation can be bypassed. In the following, we describe how we implement the attacks to bypass SMART and C-FLAT.

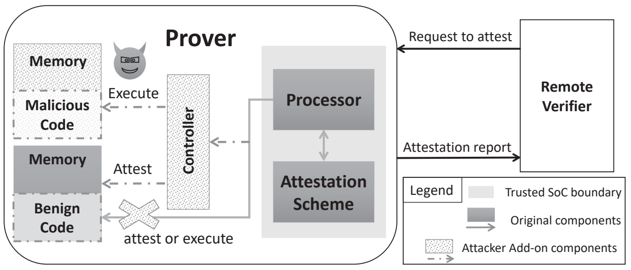


Figure 2: Memory bank attack on attestation schemes

A. SMART

SMART [9] is a static attestation scheme that establishes a root of trust in low-end embedded systems with minimal hardware components. It targets microprocessors that are able to execute code from an external memory, whereas the attestation code and key reside in an internal ROM and are protected by access control policies of a memory protection unit (MPU). When an attestation request is received, the *atomic* attestation code in ROM computes a HMAC of a region of code memory, provided in the attestation request. Then the attested code executes *atomically*.

Detecting Attestation Execution. By eavesdropping in the communication channel between the verifier and the prover for an attestation request, we determine when the attestation engine is about to run in order to launch a TOCTOU attack. Although this is permissible by the adversary model in SMART, we choose not to tackle the detection problem this way. Instead, we examine a side-channel that is inherent to the SMART design, by placing a monitor on the address bus between the processor and memory to capture which addresses are being accessed. Using the access patterns, we are able to discern whether a CPU is executing from external memory or from the internal ROM. Since SMART is prototyped on the open-source MSP430[1], it utilizes a von Neumann architecture, where data and instructions are accessed over the same address space but are structured such that they reside in different sections of memory. Hence, we can extract and filter out data accesses, leaving behind accesses to code memory. In doing so, we observe the time-frame that it takes the internal ROM to set up the attestation environment, followed by the linear scan of code addresses, then the subsequent execution of external code. On processors with modified Harvard architecture, a temporary halt in accesses to code memory would be recognized, as the ROM code starts executing. We then observe a linear scan over an address range, as code is being read and hashed by the attestation code. A break is then noticed as the ROM code cleans up memory, followed by the continued access to program memory for execution. Utilizing this, we perform one of the following attacks to mount a TOCTOU attack.

Blind Execution of Malicious Software. Since code memory remains external to the SoC, we splice the address bus, add a new memory chip containing malicious code and utilize the monitor to detect when the attestation code runs. When attesting, we bank to the memory with the intended code. When executing, we bank to the malicious code memory, allowing SMART to report valid attestation results while malicious code is actually executed by CPU during periods of no attestation.

Leakage of Secrets via Data Memory Banking. As the attestation code runs, temporary values are saved in memory, assuming SMART implementation utilizes off-chip memory to store temporary values. We use the monitor to detect when the attestation code runs. As data memory is accessed to store temporary values, we bank memories to allow for the leakage of values. We perform this by physically tampering with the address lines between the processor and the memory. As the monitor detects when SMART is about to perform its cleanup routines, we bank to a different portion of memory, leaving the ROM code to erase the wrong portion of memory. By reading the SMART secrets from memory, we are able to reconstruct the attestation secret and fake a valid response.

B. C-FLAT

C-FLAT [3] is a runtime attestation scheme that aims to measure and report the control-flow behavior of an executing code. It instruments all branch instructions such that they are intercepted by a *runtime tracer* (RTT). The RTT recovers the source and destination addresses of the branch as well as its type, which are then passed to the *measurement engine* (ME). The ME is responsible for computing a hash over the reported branches and these hash measurements are secured by running in a TrustZone secure world. In this way, a runtime control-flow attestation report is generated and verified against previously computed control-flow traces stored in a trusted verifier party.

C-FLAT is susceptible to two TOCTOU attacks assuming that the attacker has physical access to the code memory : 1) replacing instructions within a basic block with malicious ones; and 2) refactoring the control-flow graph (CFG) of an arbitrary program to match a benign CFG protected by C-FLAT. Both attacks exploit the fact that C-FLAT attests *only* control flow when exiting a basic block but not the executed instructions themselves. Hence, intermediate instructions within the basic block can be arbitrarily replaced by malicious executable code by a stronger adversary with physical access to the code

[1]http://opencores.org/project,openmsp430

memory, as long as the control flow of the code remains unchanged and the expected attestation report is not violated. These attacks are also applicable to the hardware-assisted control-flow attestation scheme LO-FAT [7] since it also only attests control flow.

We chose to implement a TOCTOU attack against one of the case studies presented in [3], namely the syringe pump program responsible for dispensing intravenous (IV) fluids. Our attack goal is to dispense liquid in incorrect volumes at unexpected times, thereby, disrupting the correct flow of IV fluids. We only demonstrate the second attack variant, however, the first variant of the attack is also easily feasible by replacing the original instructions within the basic block with malicious ones. This allows the original RTT hooks into the ME to compute a valid attestation report as it is based upon the source and destination addresses of a branch and its type.

In place of the original program that manages liquid dispensing and withdrawal, we implement a malicious program that chooses a random value to dispense by modifying the `set-quantity` function and additionally creates compound dispense and withdraw triggers for the `move-syringe` function. We embed this code in the original program, which creates new edges in the CFG of the syringe pump program. Our new edges would violate C-FLAT's attestation report for the benign syringe pump program.

To avoid triggering C-FLAT, we refactor the CFG of our attacker syringe pump program using the REpsych tool[2] to construct the desired CFG. The REpsych tool is an IDA plugin that translates a source image into a functioning program whose CFG is the image. We used the original syringe pump's CFG as a source image, and our modified syringe pump program as the target. This allowed us to generate a program with alternative functionality, but equivalent CFG to the original syringe pump program. We then recompute the attestation report using C-FLAT's tools[3]. The attacker program's attestation report matched the original syringe pump program's attestation report after CFG refactoring. Thus, we were able to accurately execute the attacker program without violating C-FLAT's protection.

IV. ATRIUM

We present ATRIUM a runtime attestation scheme targeting bare-metal embedded systems software. ATRIUM comprises a remote embedded system, called in this context the prover $\mathcal{P}rv$, and a trusted verifier $\mathcal{V}rf$. The $\mathcal{P}rv$ is deployed in-field such that the adversary has physical access to its memory. Typically, both $\mathcal{V}rf$ and $\mathcal{P}rv$ have access to the binary code of the program P to be attested on $\mathcal{P}rv$. Note that, in practice, it may not be feasible to apply runtime attestation to the entire program code due to obvious efficiency reasons, but it can be applied to pre-defined security-critical code regions.

A. *Adversary Model and Assumptions*

In addition to the standard capabilities of the adversary in typical remote attestation schemes, which assume software-only attacks, our adversary can also perform runtime attacks (§ II). Furthermore, we assume a stronger adversary that has physical access to the $\mathcal{P}rv$'s memory and can manipulate the program code at runtime and, therefore, is able to mount a TOCTOU attack (§ III). However, the adversary cannot modify memory reserved and used by ATRIUM itself – this memory is hardware-protected and not mapped to the software-accessible address space. *Data-oriented programming attacks* [13] that do not affect the control flow as well as invasive physical attacks on the SoC that aim at extracting secret keys are out of scope. This assumption is reasonable, since an adversary is more likely to mount a simple physical attack on the memory as we demonstrated in § III, rather than expensive sophisticated invasive attacks on the chip that can destruct it eventually.

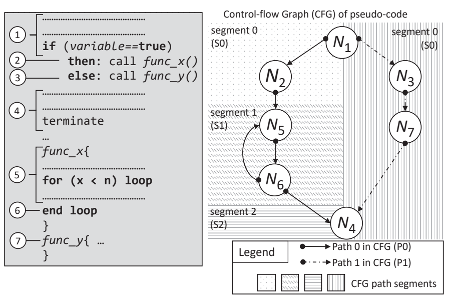
B. *Runtime Attestation: High-Level Scheme*

Inspired by C-FLAT [3] (described in § III-B) and the hardware-assisted scheme LO-FAT [7], ATRIUM performs attestation of an executing program code at runtime. However, unlike both schemes, it measures both the executed instructions (to detect the more advanced TOCTOU attacks described in § III) and control flow (to detect runtime attacks).

Similar to C-FLAT, our attestation mechanism relies on $\mathcal{V}rf$ performing one-time offline pre-processing to generate the CFG of program P (including expected loop execution information) by means of static and dynamic analysis. $\mathcal{V}rf$ computes cryptographic hash measurements over the instructions and addresses of basic blocks along legal CFG paths and stores them in a reference database. $\mathcal{V}rf$ initiates the attestation by sending $\mathcal{P}rv$ benign input in_b, the code region to be attested in P, and a nonce to ensure freshness of the attestation report. $\mathcal{P}rv$ executes P on the benign inputs in_b and potentially malicious inputs in_m that are not controlled by $\mathcal{V}rf$ and may lead to the corruption of the program's control flow by means of runtime attacks (§ II). ATRIUM is triggered during the execution of the code region of interest and computes a set of hash measurements over the executed paths. When execution of the code region is complete, $\mathcal{P}rv$ generates and sends to $\mathcal{V}rf$ the final *attestation report* consisting of the concatenated set of hash values $H_0\|...\|H_n$ and the number of iterations of the hash values which correspond to executed loop paths, and a signature over $H_0\|...\|H_n$ and the nonce based on $\mathcal{P}rv$'s secret key sk. To ensure authenticity of the report, sk is stored in memory accessible only by ATRIUM. Upon receiving the report, $\mathcal{V}rf$ verifies its signature using $\mathcal{P}rv$'s public key pk and checks whether the $H_0\|...\|H_n$ values match the reference hash values under input in_b. If they match, $\mathcal{V}rf$ concludes that $\mathcal{P}rv$'s execution of the attested code region was correct in terms of executed instructions and their control flow. For better understanding, we demonstrate next by an example how the hash values are computed during attestation.

Example. A CFG of an example pseudo-code is shown in Figure 3. Each numbered node in the CFG represents the corresponding numbered *basic block* of sequential instructions in the pseudo-code and the address of the first instruction of that basic block. For example, $\mathbf{N}_5$ corresponds to the first 3

[2]https://github.com/xoreaxeaxeax/REpsych
[3]https://github.com/control-flow-attestation/c-flat



```
1  ......................................
   ......................................
   if (variable==true)
2      then: call func_x()
3      else: call func_y()
4  ......................................
   ......................................
   terminate
   ...
   func_x{
5  ......................................
   for (x < n) loop
   ......................................
6  end loop
   }
7  func_y{ ...
   }
```

Figure 3: Example pseudo-code and its segmented CFG

instructions outlined in the pseudo-code, constituting a single basic block, and the address of the first instruction. The CFG shown in Figure 3 has 2 main paths: **P0**, in bold, consisting of nodes N_1-N_2-N_5-N_6-N_4 and **P1**, in dashed, consisting of nodes N_1-N_3-N_7-N_4. In order to avoid combinatorial explosion of legal hash values that would occur due to multiple loop iterations, the program CFG is split into segments such that hash values for loop paths are computed separately, rather than computing a single hash value over the complete executed path of the attested region. In Figure 3, due to the loop in N_5-N_6, **P0** is sectioned into 3 segments: $S0$, $S1$ and $S2$. $S0$ comprises all nodes till loop entry at N_5, where $S1$ is initialized. $S1$ ends at the loop exit node N_6, and $S2$ is initialized at N_4 and beyond until again another loop is encountered and so on.

When path **P0** is executed and attested, ATRIUM accumulates nodes (address of the first instruction and the individual instructions in each node) along each segment and computes a hash value for each segment: a hash value $H_0 = H(N_1||N_2)$ over the nodes in $S0$ of **P0**, followed by $H_1 = H(N_5||N_6)$ over the nodes in $S1$, and $H_2 = H(N_4)$ over the nodes in $S2$, resulting in the set of hash values $H_0||H_1||H_2$ representing the executed path **P0**. **P1**, on the other hand, has no loops. Therefore, when executed the whole path is measured by a single hash value $H_3 = H(N_1||N_3||N_7||N_4)$. This CFG segmentation in hash computation allows our scheme to tackle loops and nested loops efficiently, while also allowing fine-grained attestation of their execution. It requires that ATRIUM can detect and interpret loops accurately at runtime. Unlike C-FLAT, we aim to realize this without instrumentation, hence avoiding the associated performance overheads. We present next the architecture of ATRIUM and how it interfaces directly with the processor hardware to capture at runtime every executed instruction and accurately interpret control flow and infer loop entry and exit points *without instrumentation*.

V. ATRIUM: Design and Implementation

ATRIUM is a hardware-based scheme for runtime attestation that tightly integrates with a processor, as shown in Figure 4. This allows it to extract the executed instructions and their memory addresses from the execute stage of the pipeline at runtime while the program P (that needs to be attested) executes on input values in_b and in_m. ATRIUM outputs a set of hash values $H_0\|...\|H_n$ computed over the executed path

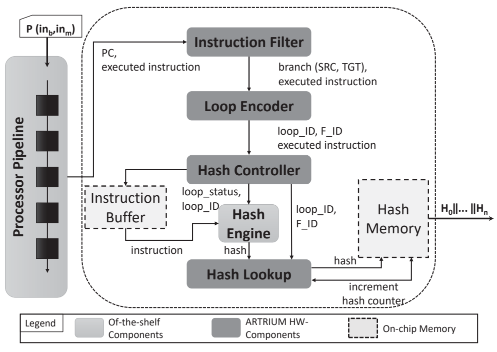


Figure 4: Architecture of ATRIUM

which get included in the attestation report. We present next the components of ATRIUM and their implementation details.

A. Instruction Filter

Upon code execution, the instruction filter extracts the current program counter (PC) and the executed instruction per clock cycle and checks whether the current instruction is a branch or jump, since such instructions reflect control-flow transitions.

Implementation. We implemented the instruction filter such that it tightly extends the execute stage of the processor from which it extracts the PC and instruction per clock cycle. If the current instruction is a control-flow instruction, its PC and the address it jumps to are stored as source–target pair, (Src, Tgt)-pair. To determine whether the branch was taken and whether control jumped forwards or backwards in memory, the PC of the next executed instruction is compared to the stored target address. Instruction filter outputs the following signals: (1) branch instructions, their type, and (Src, Tgt)-pairs and (2) basic block addresses and executed instructions.

B. Loop Encoder

As explained in § IV-B, ATRIUM handles loops and their hash computations differently. Hence, at runtime the loop encoder detects loops and identifies their entry and exit points and their depth, in case of nested loops. It checks whether the behavior of a captured branch can be inferred as returning to a loop's entry point, hence indicating a new loop iteration. The loop encoder instructs the hash controller to finalize the ongoing hash computation and initialize a new one with the entry address of a loop iteration. Furthermore, the loop encoder also detects if a branch represents a system call since system functions have to be handled specially in ATRIUM.

Implementation. To detect loops at runtime without relying on code instrumentation, we utilize a feature of RISC architectures that implement a *link-register*, such as PowerPC, ARM, SPARC, and RISC-V. We adopt a heuristic used in [7] to distinguish between backward branches that indicate loop entry, and branches for subroutine calls where the call target resides earlier in memory. Subroutine calls use instructions that update the *link-register* with the return address, hence, we consider any *non-linking* backwards branch as a *loop entry node*. Consequently, the basic block after the branch instruction is considered a *loop exit node*. This is based on observations of the RISC-V compiler assembly and its calling convention: any subroutine call with multiple call sites must be *linking* and updates the *link-register*. Subroutines with a single call site can be compiled as a *linking* branch or inlined using the RISC-V compiler. A system call is identified by comparing its target against a predefined list of addresses of such functions and issuing a unique identifier for that function F_ID. The loop encoder stores the addresses of entry and exit nodes of each loop in a content-addressable memory (CAM) to ensure single-cycle constant-access search time. At runtime, every (Src, Tgt) is used to index the CAM to detect if a loop is re-entered or exits and to extract its loop_ID and depth (in case of nested loops). An iterations counter for each loop is maintained and updated at runtime. We detect loop exit when execution proceeds past the currently active loop exit node, either due to sequential execution or a non-linking jump, such as a *break*. The F_ID, loop_ID and loop_status signals are forwarded then to the hash controller.

C. Hash Engine and Hash Controller

The hash engine computes a hash value of each executed path within a segment (§ IV-B). The hash controller regulates the operation of the hash engine, i.e., finalizes or initiates a hash computation based on the control signals received from the loop encoder. In case the computed hash corresponds to a loop path, the hash controller sends this hash to the hash lookup and sets the search boundaries of the hash lookup to that particular current loop (necessary in case of nested loops). Otherwise, the hash value is simply stored in hash memory.

Implementation. We selected Blake2 [4] for hash computations and used the open-source hardware implementation of Blake2b, which takes as an input a message block of size 1Kbit and has a configurable digest size. We configured its digest size to 88 bits to reduce memory requirements for hash lookup and hash memory. The hash controller buffers incoming instructions from the loop encoder, aligns them in 1Kbit message blocks and feeds them to the hash engine. The hash engine requires 28 cycles to process a block, thus the hash controller issues a stall signal to the processor in case its buffer is full and the hash engine cannot digest a new message block. Therefore, system calls are handled differently because we observe that they often involve short loops that are executed arbitrarily many times, e.g., string utility functions. Hashing such a short loop path every time it executes, especially for a large number of iterations, would require the hash controller to stall the processor frequently and delibitate performance. Hence, the executed instructions along a loop path are concatenated and stored in plaintext in a dedicated CAM and sent to the hash engine only once when it is first encountered. When the same path is executed again, it is compared with the previously recorded paths in the CAM, and a corresponding counter is incremented when a match is found, without sending it to the hash engine again. The counters are concatenated with the corresponding hash values in the final attestation report.

[4]https://blake2.net/

Upon finalizing a hash computation, the hash controller checks, whether the resulting hash is computed over a path within a loop or not. If it is computed over a path loop, it forwards the resulting hash value from the hash engine synchronized with its corresponding loop_ID to the hash lookup.

D. Hash Lookup

The hash lookup is dedicated to storing and tracking hash values during loop iterations efficiently. Once a hash value is ready, the hash controller forwards it to the hash lookup, which searches within the current loop's list of hash values for a match. If not found, then the hash value is appended to the list. The hash lookup also maintains a counter per loop path which is incremented when its corresponding hash is encountered.

Implementation. To avoid multiple memory accesses due to sequential search of a particular hash value, we implement the hash lookup as a set of CAMs, whose number can be configured based on the system's requirements. A CAM is dedicated for every active loop, so the number of CAMs is determined by the maximum number of nested loops that ATRIUM is configured to track concurrently. Each CAM has a configurable capacity of (n,m) bits, where n is the maximum number of entries and m is number of bits per entry and a counter to maintain the occupied number of entries. When a loop is detected, the hash controller sends the hash lookup to reserve a CAM for it and reset its counter to zero. The CAM holds the computed hash values of a currently executing loop temporarily till the loop exits. Each time a path in the pertinent loop is executed, its computed hash value is looked up in the associated CAM. If a match is not found, i.e., this path has not been executed before, then its hash value is appended to the CAM. When a new loop is detected and all CAMs are occupied, a CAM that was reserved for a loop that already exit (and will not be executed again) is freed and re-used. If a path does not belong to a loop, then its hash value is used to update the hash memory directly.

E. Hash Memory

All computed hash values are stored in a dedicated memory. After the execution of the code region to be attested completes, these hash values are assembled and a digital signature is computed over them. The hash values $H_0\|...\|H_n$ and their signature are then transmitted to $\mathcal{V}rf$.

Implementation. An on-chip hash memory is dedicated to store all computed hash values during a single attestation run of the pertinent code region. The sequence of the storage of the hash values in memory indicates the order of the first occurrence of their corresponding code segments during execution. It is necessary to maintain this order and report $H_0\|...\|H_n$ in the same sequence to $\mathcal{V}rf$ for correctly verifying execution. In our FPGA prototyping of ATRIUM (cf. § VI), we configure the hash memory as on-chip block RAM (BRAM) of configurable capacity with each entry occupying 88 bits for hash digest and 8 bits for its counter. The capacity is configured according to our attestation requirements, i.e., the maximum number of CFG segments an attested code region would consist of. Alternatively, for constrained embedded systems, we can reduce the memory requirements by streaming the hash values (or every batch of them) as soon as they get generated to the $\mathcal{V}rf$.

VI. Evaluation & Security Considerations

A. Performance & Area Evaluation

We implemented ATRIUM in Verilog, interfaced it with the open-source RISC-V Pulpino core [5], and simulated and synthesized it. Performance and functionality were evaluated using a suite of microprocessor benchmarks including *Dhrystone*, *mt-matmul*, *rsort*, *spvm* and *towers*.

Functionality. We extended the Pulpino RTL with ATRIUM and performed cycle-accurate simulation on ModelSim while executing the aforementioned benchmarks. We confirm correct functionality of ATRIUM by comparing simulation results with reference execution profiles of the benchmarks, which we extracted by running the benchmarks on standalone Pulpino without ATRIUM and analyzing the execution trace.

Area and Memory. Area utilization depends on the configurations of the hash lookup and hash memory of ATRIUM. For our evaluation, we configured the hash lookup with 8 CAMs, each CAM with $n = 8$ entries and each entry being $m = 88$ bits. This allows ATRIUM to track up to 8 active nested loops at once with a maximum of 8 different $88 - bit$ path hashes per loop. On synthesizing ATRIUM using Xilinx Vivado on a Zedboard (Virtex-7 XC7Z020 FPGA), we show the overall area utilization to be 15% of slice registers and 20% of slice LUTs of this FPGA, while only one 18Kbit BRAM is required for the hash memory.

Performance. Implementation results indicate that ATRIUM can operate at a maximum clock frequency of 70 MHz on a Zedboard (Virtex-7 xc7z020 FPGA) and is, hence, on par with the Pulpino's maximum clock frequency of 50 MHz on the same board. Performance experiments show an overhead of 1.97% for *Dhrystone*, 12.23% for *mt-matmul*, 22.69% for *rsort*, 6.06% for *spvm* and 1.7% for *towers*. Since ATRIUM components run on par with Pulpino, performance loss is caused by the hash function, as the processor stalls occur *only* when the currently executed path has ended and needs to be hashed while the hash engine is still processing the previously executed path and is not ready for input. This overhead is incurred for loops with paths whose number of executed instructions are less than the required number of cycles for the hash engine to finalize its computation (28 cycles for the chosen hash function). To mitigate this overhead, the hash engine should be clocked at a higher frequency than the processor if possible.

B. Security Considerations.

We assume that the used cryptographic primitives are secure. Upon receiving an attestation request, $\mathcal{P}rv$ generates and sends the list of computed hash values $H_0\|...\|H_n$ along with a digital signature computed over it and a nonce provided by $\mathcal{V}rf$ and signed by $\mathcal{P}rv$'s secret key sk. The signature guarantees the authenticity of the attestation report while the nonce ensures its freshness. By verifying the signature, checking the value of the nonce, and comparing the received hashes to their expected values stored in $\mathcal{V}rf$'s database, $\mathcal{V}rf$ gains assurance of the correct execution (both instruction and their control flow) of the current program on $\mathcal{P}rv$. We consider three classes of attacks that can be mounted on ATRIUM.

Malware and Network Attacks. ATRIUM detects malicious software modification introduced by the adversary, as every executed instruction is included in the hash computation. To evade detection, finding a second image that maps to same hash value is required. However, that is infeasible since the hash engine is second pre-image resistant. Forging the signature or replaying an old signature is also not feasible, due to security of signature scheme and to the nonce being long enough.

Runtime Attacks. Since basic block addresses are included in hash computations along with the executed instructions, the hash values computed in ATRIUM reflect the control flow of the executed path. Being tightly integrated with the processor, ATRIUM is guaranteed to track and record every control-flow event executed. An attacker who knows the program code P or CFG(P) can try to bypass ATRIUM by searching for a second pre-image of the corresponding hash. However, by using cryptographically-secure hash function, finding collisions is computationally infeasible.

Physical Attacks. An adversary with physical access to $\mathcal{P}rv$ can try to manipulate the program code in $\mathcal{P}rv$'s memory at runtime, i.e, between time of attestation and time of execution. However, in ATRIUM attestation is performed *during* execution. Therefore, it is guaranteed that *every instruction* that is executed on $\mathcal{P}rv$ will be included in the hash generation, and consequently any manipulation will be detected by $\mathcal{V}rf$, as the generated hash values would not match $\mathcal{V}rf$'s expectations. This defends against TOCTOU attacks that can occur when attestation is *followed* by execution, as was the case for both SMART [9] and C-FLAT [3]. Finally, fault injection attacks which target the SoC clock and cause unintended behavior would also be detected by $\mathcal{V}rf$, as long as the attacks affect the instructions executed or their control flow. Note that, expensive invasive/semi-invasive physical attacks on the SoC are considered out of scope in this work.

VII. Related Work

Attestation Schemes. Existing static attestation schemes such as software-based [14], [20], hardware-based [21], [17], and hybrid [15], [9] attestation schemes are vulnerable to runtime attacks. Control-flow attestation (C-FLAT) aims at enhancing the security of static attestation schemes by additionally hashing the code's execution control flow. This enables the detection of code-reuse and non-control data attacks that divert the execution flow. However, due to frequent hash calculations and context switching (on TrustZone), C-FLAT incurs high performance overhead. LO-FAT [7] leverages hardware assistance to track and measure control flow, thus, overcoming the limitations of C-FLAT and enabling efficient attestation of *uninstrumented* code. LO-FAT, however, incurs significant area overhead due to its on-chip memory requirements (up to 49 36Kbit Block RAMs are used sparsely to store counters of

[5]https://github.com/pulp-platform/pulpino

loops' paths). Finally, in a stronger adversary model with physical access to the prover's device, these schemes are vulnerable to Time of Check Time of Use (TOCTOU) attacks. ATRIUM mitigates this by providing both static and control-flow attestation in a stronger (and more realistic) adversary model efficiently.

Authenticated Memory Modules. Authenticated Memory Modules (such as Intel Authenticated Flash [1]) aim at resisting physical attacks on external memory by preserving the memory's integrity. However, they are insecure under an adversary model with physical access. Moreover, they do not authenticate the control flow of the execution. On the contrary, ATRIUM provides an additional defense against software runtime attacks by coupling the attestation of both the instructions and their control flow with their execution to eliminate any room for TOCTOU attacks.

Memory Authentication. Such schemes [8], [6] aim at resisting physical attacks on external memory. However, they incur high performance overhead by authenticating memory blocks before execution and are susceptible to runtime attacks. ATRIUM detects both runtime attacks and physical attacks on code memory while incurring minimal overhead.

Hardware Security Architectures. Finally, hardware security architectures (such as Intel SGX) provide memory authentication as well as static attestation. However, such architectures are not designed to target low-end embedded devices. Furthermore, they only provide static attestation and therefore cannot meet the goals that we target. Nevertheless, they provide security features complementary to our work.

VIII. Conclusion

Due to the ubiquity of interconnected embedded systems, software running on these devices have become vulnerable to remote software attacks. Previous attestation schemes have been proposed to detect these attacks while always ruling out physical attacks. In this paper, we showed that physical attacks on the system's code memory are indeed feasible. We presented a hardware-based efficient scheme ATRIUM that allows precise attestation of both executed instructions as well as their control flow. ATRIUM is the first attestation scheme to provide security guarantees against a stronger adversary with physical access to code memory, and does not require any code instrumentation (compliant to legacy software) or instruction set extension. Our proof-of-concept implementation is highly efficient with reasonable performance impact on the attested software at an expense of minimal area overhead and memory.

Acknowledgments. We thank the authors of SMART for sharing their scheme. This work was supported by the German Science Foundation CRC 1119 CROSSING, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant No. 643964 (SUPERCLOUD), the U.S. Department of Energy through the Early Career Award (DE-SC0016180), the National Science Foundation Graduate Research Fellowship Program under grant No. 1144246, and the Intel Collaborative Research Institute for Secure Computing (ICRI-SC).

References

[1] Intel Authenticated Flash. www.design-reuse.com/articles/16975.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. "Control-flow integrity: Principles, implementations, and applications". *ACM Transactions on Information and System Security*, 2009.

[3] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. "C-FLAT: Control-flow attestation for embedded systems software". In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. "When good instructions go bad: Generalizing return-oriented programming to RISC". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. "Non-control-data attacks are realistic threats". In *USENIX Security Symposium*, 2005.

[6] R. de Clercq, R. De Keulenaer, P. Maena, B. Preneel, B. De Sutter, and I. Verbauwhede. "SCM: Secure code memory architecture". In *ACM Symposium on Information, Computer and Communications Security*. ACM, 2017.

[7] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. "LO-FAT: Low-overhead control flow attestation in hardware". In *Design Automation Conference*, 2017.

[8] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres. "Hardware mechanisms for memory authentication: A survey of existing techniques and engines". In *Transactions on Computational Science IV*. 2009.

[9] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito. "SMART: Secure and minimal architecture for (establishing dynamic) root of trust". In *Annual Network and Distributed System Security Symposium*, 2012.

[10] A. Francillon and C. Castelluccia. "Code injection attacks on Harvard-architecture devices". In *ACM SIGSAC Conference on Computer and Communications Security*, 2008.

[11] V. Haldar, D. Chandra, and M. Franz. "Semantic remote attestation: A virtual machine directed approach to trusted computing". In *Virtual Machine Research And Technology Symposium*, 2004.

[12] Hewlett-Packard. "Data execution prevention", 2006.

[13] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-oriented programming: On the effectiveness of non-control data attacks". In *IEEE Symposium on Security and Privacy*, 2016.

[14] R. Kennell and L. H. Jamieson. "Establishing the genuinity of remote computer systems". In *USENIX Security Symposium*, 2003.

[15] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. "TrustLite: A security architecture for tiny embedded devices". In *ACM SIGOPS EuroSys*, 2014.

[16] T. Kornau. "Return oriented programming for the ARM architecture". Master's thesis, Ruhr-University Bochum, 2009.

[17] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. "New results for timing-based attestation". In *IEEE Symposium on Security and Privacy*, 2012.

[18] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. "Code-pointer integrity". In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[19] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. "SoK: Automated software diversity". In *IEEE Symposium on Security and Privacy*, 2014.

[20] Y. Li, J. M. McCune, and A. Perrig. "VIPER: Verifying the integrity of peripherals' firmware". In *ACM SIGSAC Conference on Computer and Communications Security*, 2011.

[21] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. "Copilot – A coprocessor-based kernel runtime integrity monitor". In *USENIX Security Symposium*, 2004.

[22] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. "Design and implementation of a tcg-based integrity measurement architecture". In *USENIX Security Symposium*, 2004.

[23] H. Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In *ACM SIGSAC Conference on Computer and Communications Security*, 2007.

[24] L. Szekeres, M. Payer, T. Wei, and D. Song. "SoK: Eternal war in memory". In *IEEE Symposium on Security and Privacy*, 2013.

F CHASE: CONFIGURABLE HARDWARE-ASSISTED SECURITY EXTENSION FOR REAL-TIME SYSTEMS

[6] Ghada Dessouky, Shaza Zeitouni, Ahmad Ibrahim, Lucas Davi, Ahmad-Reza Sadeghi. "CHASE: Configurable Hardware-Assisted Security Extension for Real-Time Systems". In Proceedings of the 38th ACM/IEEE International Conference on Computer Aided Design (ICCAD'19), 2019.

CHASE: A Configurable Hardware-Assisted Security Extension for Real-Time Systems

Ghada Dessouky*⋆, Shaza Zeitouni*⋆, Ahmad Ibrahim*, Lucas Davi† and Ahmad-Reza Sadeghi*
*Technische Universität Darmstadt, Germany, {ghada.dessouky, shaza.zeitouni, ahmad.ibrahim, ahmad.sadeghi}@trust.tu-darmstadt.de
†Universität Duisburg-Essen, Germany, lucas.davi@uni-due.de

***Abstract*—Real-time autonomous systems are becoming pervasive in many application domains such as vehicular ad-hoc networks, smart factories and delivery drones. The correct functioning of these real-time systems is timing-critical with hard deadlines. However, although they interact with other systems and exchange inputs/outputs with the physical world, they usually lack security mechanisms, which makes them susceptible to a wide range of attacks with critical consequences. Typically, this is because security mechanisms usually violate the real-time requirements of these systems and cannot be adjusted at runtime to provide the adequate security without compromising performance.**

In this paper, we propose a consolidated runtime-configurable hardware-assisted security extension called CHASE that supports different levels of security at runtime. Depending on the desired security level and the system real-time, availability or functionality requirements, CHASE can be configured accordingly at runtime, thus enabling the calibration of the security vs. performance trade-off. We analyze CHASE's effectiveness in providing different security guarantees against various adversarial capabilities, and show how this is achieved with reasonable logic overhead and minimal performance overhead.

***Index Terms*—Remote attestation, real-time system security, runtime attacks, hardware-assisted security, runtime attestation**

I. Introduction

Real-time systems are ubiquitous in many application domains such as programmable logic controllers (PLCs), electronic control units (ECUs) and emerging domains that deploy networks of collaborative autonomous systems, e.g., vehicular ad-hoc networks, smart factories, search and rescue, and delivery robots and drones. Typically, such systems are required to perform their tasks in real time, while some may have hard and critical time deadlines with little tolerance for down time. They may also be deployed in safety-critical or non-deterministic infrastructures where their fail-safe operation is paramount. To perform their tasks, they might also be interconnected with the physical world and other devices, making them equally vulnerable as other systems to security exploits.

Security for Timing-Critical Applications. Nevertheless, these systems usually lack security protection mechanisms, leaving them exposed to a wide spectrum of attacks, such as the infamous Stuxnet[1] and more recently Triton[2]. Particularly, an attacker may violate memory integrity by exploiting a standard memory corruption vulnerability, e.g., externally-controlled format string[3] that causes buffer overflows leading to data memory corruption. By corrupting targeted control-flow information stored in the stack or the heap and overwriting code-pointers (return addresses or function pointers), an attacker can redirect the control flow of execution to cause a malicious and unauthorized effect. Such *runtime attacks* can be used to inject malicious code (code-injection attacks) or re-use already existing benign code chunks maliciously (code-reuse attacks), such as control-flow hijacking and data-oriented attacks [1], [2].

Protection mechanisms, such as control-flow integrity (CFI) [3] and control-flow attestation [4]) to mitigate or detect such attacks have been shown to incur non-negligible performance overheads. While this can be tolerated to some extent for applications without real-time constraints, it would violate the functionality requirements of real-time high-availability systems. Even defenses such as asynchronous CFI specifically designed for PLCs [5] still incur performance overhead of up to 8.3%. According to the NIST 800-82 guide on the security of industrial control systems [6], timing, safety and availability requirements must be prioritized when designing security mechanisms for these systems.

Attack Space Coverage and Reconfigurability. Moreover, existing defenses against runtime attacks each assume a particular adversary model and thus mitigate specific classes of attacks. Currently, no consolidated defense exists that can mitigate multiple classes of different attack vectors, or can be at least configured to thwart different adversarial capabilities depending on the desired security requirements and deployment environment. This is especially true for hardwired hardware-assisted security extensions [7]–[12] which cannot be upgraded or patched after fabrication. This makes system architects reluctant to deploy them, despite their advantages over software-based defenses.

Our Goal. In this work, we aim to tackle the challenges outlined with respect to attack space coverage and applicability of defenses for timing-critical applications. These challenges hinder the practical deployment of hardware-assisted security mechanisms for embedded systems in general, and for real-time safety-critical systems in particular. In doing so, we address the persistent trade-off between functionality requirements (e.g., real-time operation, safety, availability or other deployment constraints) vs. security requirements. We aim to provide a flexible means for the system designer to tune this trade-off by only incurring the corresponding performance overhead for the degree of the security guarantees required and configured at runtime.

To achieve this, we categorize the different classes of attacks that may target embedded systems. We evaluate which of these attacks can be detected on-device and which of them require more sophisticated policy-checking at a trusted third party. With this in mind, we provide a consolidated and configurable defense mechanism that can be adjusted at runtime to provide different security services (and thus different security guarantees against different classes of attacks) at the cost of different performance overheads. We enable this by leveraging a custom hardware-based extension, called CHASE, designed to operate in parallel to the actual processor. It captures and tracks the execution at runtime in a cycle-accurate and tightly coupled manner. CHASE is runtime-configurable and supports different security services to mitigate different adversarial capabilities, where

⋆Authors contributed equally to this work

[1]https://www.mcafee.com/enterprise/en-us/security-awareness/ransomware/what-is-stuxnet.html

[2]https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/triton-malware-spearheads-latest-generation-of-attacks-on-industrial-systems/

[3]CWE-134: Use of Externally-Controlled Format String https://cwe.mitre.org/data/definitions/134.html

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

its hardware can be configured at runtime to either *verify* or *enforce* control flow (including call-return matching). Verifying control flow checks on-device that a control-flow transfer is allowed *after* it is executed, without incurring performance overhead on the execution. Conversely, enforcing control flow actively checks that each control-flow transfer is allowed *prior* to its execution, which evidently comes with some performance overhead. For more sophisticated attacks that cannot be detected on-device, CHASE can be configured to capture fine-grained features of the relevant execution, and send them to a trusted third party (equipped with computational resources) for verification using more complex policies.

For timing-critical systems, CHASE can be configured to verify on-device that the control flow of execution is valid without interfering with the application run time or halting the execution in case of illegal control flow. Violation of a control flow policy is detected with minimal latency, after which CHASE reacts gracefully without compromising the safety or real-time requirements of the application. Execution is redirected to a pre-defined and isolated safe state which is application-dependent. In the meantime, a neighboring device (in case of a network of collaborative devices) and the trusted third party are notified of the impending exploit to react accordingly.

Contributions. In this work, we tackle the challenges we outline above by proposing:

- A modular hardware-assisted extension CHASE that consolidates different defenses and security services mitigating a larger attack space than existing defenses.
- Configurable security that can be adjusted at runtime depending on the security vs. functionality requirements of a given embedded application and the deployment scenario.
- A security mechanism that can verify control flow on-device with minimal latency making it suitable for timing-critical systems (with caveats).
- Proof-of-concept implementation for a RISC-V processor and an evaluation of performance overhead for the detection latency.

II. CHASE: System and Adversary Model

The intuition behind our work is two-fold. The first is that security mechanisms to date are *not configurable by design* to provide different levels of security at runtime. Configurable security is required when an embedded system is deployed in scenarios with different threat levels and different functionality requirements (fail-safety, real-time, periodicity, etc.). This is particularly a limitation of existing hardware-assisted extensions which cannot be modified or upgraded after production, rendering them hardwired to mitigate a fixed class of security threats for the entire lifetime of the encompassing system. Despite their many advantages as opposed to software-based schemes, this hinders their deployment in practice.

The second is that existing security extensions, even asynchronous Control-Flow Integrity (CFI) designed for PLCs [5], affect the application run time, often non-deterministically. This makes them unsuited for timing-sensitive systems that have no tolerance for down time or variable reaction times. This further emphasizes the need for a tuning knob that can be used to calibrate the trade-off between the security guarantees vs. application run time and performance overhead for different use cases and deployment settings.

To tackle the above challenges and provide configurable security and modular lines of defense, we propose CHASE. CHASE is the first consolidated and configurable custom hardware-assisted security extension that integrates tightly with the processor core of a remote in-field mid-end embedded device, called $\mathcal{DEV}$. CHASE captures and tracks the execution of $\mathcal{DEV}$ at runtime in a cycle-accurate manner using custom hardware that operates in parallel to the processor execution. It supports different security configuration modes where valid control flow can be either actively-enforced or attested after-the-fact, assuming control-flow policies are provisioned on $\mathcal{DEV}$. Verification can be performed either *on-device* or *remotely*. On-device verification checks control-flow transfers against control-flow policies that are provisioned on $\mathcal{DEV}$. Although on-device verification is limited to detecting explicit control-flow hijacking, it is low-latency and can mitigate an exploit within a few clock cycles as shown in § V-B.

Remote attestation, on the other hand, is used to detect more sophisticated attacks, such as non-control-data attacks, which do not directly compromise the control flow. It requires that the recorded and measured execution is reported to a trusted third party (called the moderator $\mathcal{MOD}$). $\mathcal{MOD}$ is assumed to be a significantly more computation-resourceful server than $\mathcal{DEV}$ and, thus can verify the reported execution against a more complex set of policies and heuristics, and detect data-oriented attacks. At runtime, the user can select to activate or deactivate any of these security services depending on the trade-off between the functionality requirements (e.g., how much performance overhead can be tolerated) and the presumed threat level of the deployment settings. In what follows, we describe and classify the adversarial capabilities and classes of attacks we consider in this work.

A. Adversary Model and Assumptions

Adversary Model. We consider an adversary $\mathcal{ADV}$ with varying capabilities depending on the deployment settings, and with full control over both the *program memory* and *data memory* of the target program executing on $\mathcal{DEV}$.

The different types of attacks $\mathcal{ADV}$ can launch against embedded systems can be broadly classified into:

A1 Static code manipulation (malware injection) attacks
A2 Runtime code-injection attacks
A3 Runtime control-flow attacks
A4 Runtime non-control-data attacks
A5 Runtime code manipulation attacks

$\mathcal{ADV}$ can launch static code manipulation attacks and inject malware such that modified code is loaded at start-up and executed (**A1**). $\mathcal{ADV}$ can also launch runtime attacks (**A2** - **A5**) by exploiting memory corruption vulnerabilities that cause buffer overflows leading to corruption of data memory. By corrupting control-flow information stored in the stack or heap and overwriting code-pointers (return addresses and function pointers), $\mathcal{ADV}$ can maliciously redirect the control flow of execution at runtime. In *code-injection* attacks (**A2**), the attacker places a malicious executable payload in program memory and redirects control flow to execute it. Alternatively, state-of-the-art runtime attacks exploit *code-reuse* techniques, e.g., *Return-Oriented Programming* (ROP) [2]. These attacks exploit memory corruption vulnerabilities and stitch together benign *gadgets* of code, which already reside in the program memory, in a particular sequence to build the attack payload and hijack the control flow of the program maliciously. These attacks hijack the control flow of the program by executing invalid control-flow transfers, that do not exist in the control-flow graph (CFG) of the program (**A3**).

More sophisticated code-reuse attacks known as non-control-data attacks (**A4**) do not explicitly compromise the control flow of a program, but cause malicious execution by corrupting critical data variables such as an authentication variable or loop variable. This results in executing a privileged (unintended) but permissible control-flow path that exists in the CFG or manipulating the number of iterations of a program loop or control-flow edge. This can have severe consequences depending on the context and is more challenging to detect. Furthermore, a stronger $\mathcal{ADV}$ can modify program code in memory at runtime through *physical access* without mounting sophisticated invasive physical attacks. $\mathcal{ADV}$ can replace the benign code memory with malicious code memory at runtime especially if the program code resides in an external off-chip memory [10] (**A5**).

Assumptions. We assume that $\mathcal{MOD}$ has access to the source and binary code of the target program and that static root of trust is in place by deploying conventional static (*binary*) attestation at load-time. This assures that $\mathcal{DEV}$ is executing unmodified program code, thus effectively mitigating attacks **A1**. Static attestation is a standard established mechanism assumed to be commonly deployed in embedded systems, while incurring no overhead on the application run time. Access to the source code is assumed for $\mathcal{MOD}$ to generate the control-flow graph of the target program and define the set of policies to be enforced/attested (described in more detail in III-A). Code injection attacks **A2** can be effectively prevented by marking memory as either writable or executable (W$\oplus$X). This mechanism, known as *Data Execution Prevention* (DEP) [13], is long-established and considered a standard assumption for these systems. Finally, we assume that expensive invasive physical attacks are out of scope (except attacks **A5** because they are realistic in practice). Thus, $\mathcal{ADV}$ cannot compromise hardware-protected memory used exclusively by CHASE that is not mapped to the software-accessible address space.

B. Requirements Analysis

To address the above, we derive the following requirements:

R1 *Configurable security:* Different security services with varying security guarantees should be supported. These should be configured at runtime depending on the threat level presumed for the deployment and functionality requirements.

R2 *Runtime security:* The security services should be capable of detecting different classes of runtime attacks (**A3** - **A5**) when activated. For at least one of these services, the attacks should also be detected with a sufficiently low latency for an impending exploit to be prevented in time (not just after-the-fact detection).

R3 *Minimal performance impact:* All services supported should incur minimal performance overhead on $\mathcal{DEV}$. At least one of these services should deterministically guarantee zero performance overhead and no impact on the application run time.

R4 *Accuracy & completeness:* All services, when enabled, should accurately capture, record and enforce or attest every control-flow event in the execution. No control-flow events can be dropped or bypass the activated security mechanism.

R5 *Secure communication:* Whenever applicable, attestation results should be securely reported to $\mathcal{MOD}$; they should be integrity-protected and fresh.

R6 *Reasonable logic overhead:* The hardware extension providing these security services should incur minimal memory and logic overhead to the baseline processor.

III. CHASE: High-Level Design

In light of the requirements described in § II, we present CHASE, the first consolidated hardware-assisted security extension that is *configurable by design* in the face of different attack classes § II-A based on their degree of difficulty to launch and (effectively) detect/mitigate. With this in mind, we describe the different security configuration modes supported in CHASE to mitigate these attacks and how they can be configured at runtime in § III-A. These modes aim to provide varying security guarantees by means of different security mechanisms to thwart different adversarial capabilities at different performance overhead costs. In § III-B, we describe the modular design of CHASE shown in Figure 2 and how its components can be configured to realize the different configuration modes at runtime.

Attack Categorization. As described in § II-A, attacks **A1** and **A2** are the most trivial to launch and mitigate. The former are mitigated by deploying static attestation at load-time. The latter are mitigated by deploying Data Execution Prevention (DEP), a long-established mechanism that does not affect application run time. Attacks **A3**, **A4**, and **A5** are the more challenging to launch and mitigate, and are currently the more sophisticated threats targeting embedded systems. CHASE provides different modes of configuration for different security guarantees against these attacks, thus fulfilling **R1** in § II-B. This enables the configuration of different security mechanisms at runtime while having the targeted application, functionality requirements and the threat level in mind, thus calibrating the performance/security trade-off flexibly. For real-time applications, for instance, particular configurations can be selected that do not affect the application run time, while providing an adequate level of security. For more vulnerable or less timing-critical applications, higher security guarantees can be provided by enabling other mechanisms, but at a higher performance overhead. This renders CHASE suitable for deployment in a wide spectrum of embedded systems with different use cases, while also taking into account the strict functionality requirements of timing-critical systems. We describe next how these different security services can be activated at runtime.

A. Security Configurations Scheme

Four configuration modes are supported by CHASE:

C1 On-device control flow verification
C2 On-device control flow enforcement
C3 Moderator-verified control flow attestation
C4 Moderator-verified executed instructions attestation

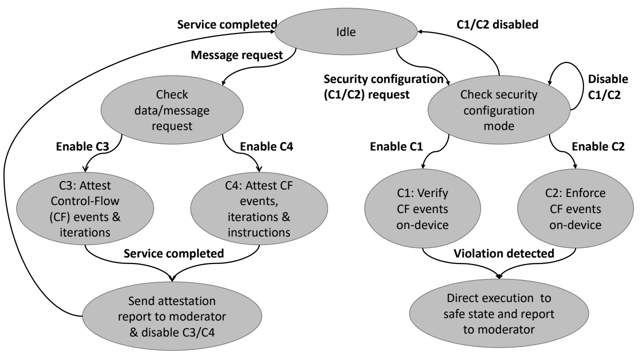

Idle
Service completed
C1/C2 disabled
Message request
Security configuration (C1/C2) request
Disable C1/C2
Check data/message request
Check security configuration mode
Enable C3
Enable C4
Enable C1
Enable C2
C3: Attest Control-Flow (CF) events & iterations
C4: Attest CF events, iterations & instructions
C1: Verify CF events on-device
C2: Enforce CF events on-device
Service completed
Violation detected
Send attestation report to moderator & disable C3/C4
Direct execution to safe state and report to moderator

Fig. 1: Runtime activation of the CHASE configuration modes.

To enable these security mechanisms, a set of reference control-flow policies is generated and provisioned in a dedicated addressable policy memory on $\mathcal{DEV}$, while more complex policies are made available at $\mathcal{MOD}$. These policies are generated by means of offline one-time static and dynamic code analysis. The addresses used to access the policy memory for fetching the policies for indirect branch instructions are instrumented directly after the corresponding indirect jump instructions in the application binary (or source). Moreover, code analysis is used to generate a priori the list of security-critical data/messages that can be requested from $\mathcal{DEV}$, and the

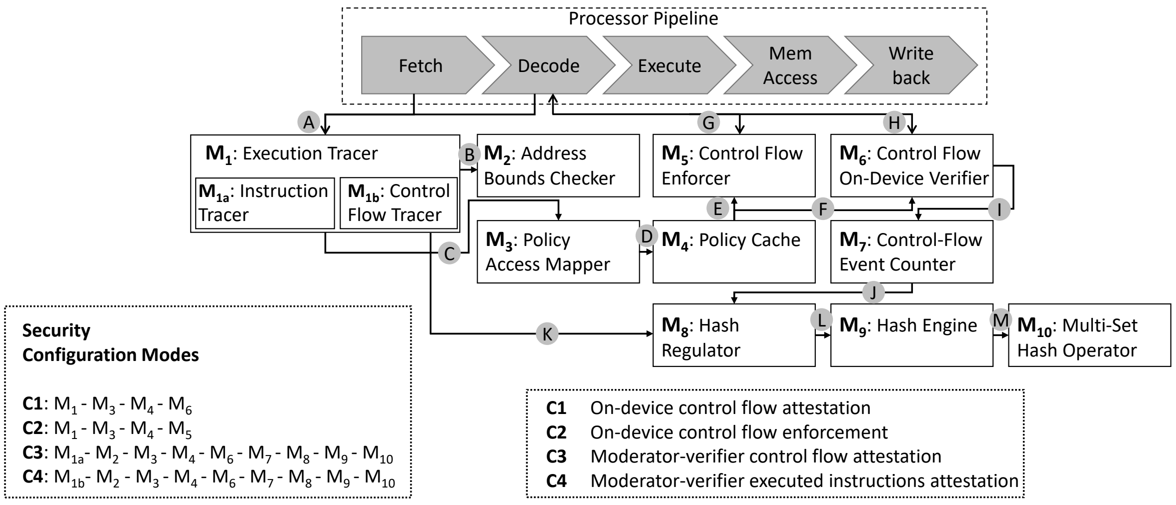

Processor Pipeline
Fetch
Decode
Execute
Mem Access
Write back
A
B
C
D
E
F
G
H
I
J
K
L
M
M_1: Execution Tracer
M_{1a}: Instruction Tracer
M_{1b}: Control Flow Tracer
M_2: Address Bounds Checker
M_3: Policy Access Mapper
M_4: Policy Cache
M_5: Control Flow Enforcer
M_6: Control Flow On-Device Verifier
M_7: Control-Flow Event Counter
M_8: Hash Regulator
M_9: Hash Engine
M_{10}: Multi-Set Hash Operator
Security Configuration Modes
C1: M_1 - M_3 - M_4 - M_6
C2: M_1 - M_3 - M_4 - M_5
C3: M_{1a}- M_2 - M_3 - M_4 - M_6 - M_7 - M_8 - M_9 - M_{10}
C4: M_{1b}- M_2 - M_3 - M_4 - M_6 - M_7 - M_8 - M_9 - M_{10}
C1 On-device control flow attestation
C2 On-device control flow enforcement
C3 Moderator-verifier control flow attestation
C4 Moderator-verifier executed instructions attestation

Fig. 2: CHASE high-level hardware architecture representing the modules activated for different security configuration modes.

corresponding software modules that contribute to the generation of the pertinent data/message, and their memory address bounds. This list is provisioned in a dedicated memory on $\mathcal{DEV}$ along with the data/message ID, and is used to restrict the attestation in modes **C3** and **C4** to the relevant software modules only, when a particular data/message is requested as described below.

Modes **C1** and **C2** can be enabled or disabled at any point during execution as shown in Figure 1, and are not bound to the execution of a particular data/message request. They verify or enforce, respectively, that the control flow of execution is valid so long as they are enabled. Modes **C3** and **C4**, on the other hand, are bound to a particular data/message request. If either mode is requested along with the data/message request, the attestation service continues to run until the request is completed. We describe next how the hardware modules of CHASE shown in Figure 2 are configured at runtime to realize the different configuration modes.

B. Security Configuration Modes

C1. Control-flow transfers are captured at runtime by module $\mathbf{M}_{1b}$ in Figure 2 while being executed. The captured control-flow events are checked against control-flow policies, which enlist the allowed destination addresses for every indirect branch instruction. These policies are in a dedicated policy memory and are pre-fetched into a dedicated on-chip cache (policy cache). The required policy address is mapped to the corresponding cache entry by module $\mathbf{M}_3$ and used to fetch the policies from cache in $\mathbf{M}_4$. $\mathbf{M}_6$ verifies the captured control-flow transfer by comparing it against the fetched policy if it is a forward transfer, e.g, a function call. The call site address is then pushed to the call-return matching stack in $\mathbf{M}_6$ if this is a `jump-and-link-register` instruction (function call). Backward control-flow transfers, e.g., returning to a call site, are verified by comparing the return address with a pre-defined number of most-recently call sites pushed onto the stack. The matching call site address and addresses pushed on top of it afterwards are all popped off the stack. The activated paths for this mode are A, C, D, F, and H through the activated modules $\mathbf{M}_{1b}$, $\mathbf{M}_3$, $\mathbf{M}_4$, and $\mathbf{M}_6$ as shown in Figure 2. This mode guarantees low-latency (within a few clock cycles) detection of illegal control-flow transfers and graceful mitigation of an impending exploit (**A3** attacks) while incurring zero performance overhead on the application run time (with caveats that we discuss in § V-B and § VI).

C2. A control-flow transfer is *not permitted* to occur unless it is validated on-device by $\mathbf{M}_5$ in this mode. This invasive enforcement of control-flow transfers guarantees prevention of executing illegal control-flow transfers (**A3** attacks) that violate the provisioned policies, while incurring minimal performance overhead. This overhead is variable and proportional to the number of control-flow transfers and the number of policy rules verified for each control-flow transfer. The activated paths for this mode are A, C, D, E, and G through the activated modules $\mathbf{M}_{1b}$, $\mathbf{M}_3$, $\mathbf{M}_4$, and $\mathbf{M}_5$.

C3. To detect more sophisticated control-flow and non-control-data attacks (**A3** and **A4** attacks), a more detailed attestation is performed that cannot be verified on-device owing to the complexity of the required code analysis and derived policies. Since this mode is activated for a specific data/message request, the current instruction address is checked by $\mathbf{M}_2$ to validate whether it is within the memory bounds of relevant program modules that should be attested for this particular data/message. This avoids attesting modules that may execute in parallel but are irrelevant to the requested data/message. If a control-flow event is within bounds, i.e., to be attested, its policy address is fetched from cache by $\mathbf{M}_3$ and $\mathbf{M}_4$ (similar to **C1** and **C2**). Then the control-flow event is verified on-device by $\mathbf{M}_6$ and the counter for this particular edge is incremented by $\mathbf{M}_7$ to keep track of the number of times a control-flow edge executes. Each executed edge is measured into a cryptographic hash computation by the hash engine ($\mathbf{M}_9$) and forwarded to the multi-set hash operator ($\mathbf{M}_{10}$). Both are regulated by ($\mathbf{M}_8$). The multi-set hash (MSH) function enables the computation of a single fixed-length hash digest for a set of elements (control-flow edges in this case) while allowing its members to occur multiple times, in which the order of the items does not affect the final value [14]. The MSH functionality significantly reduces the amount of information that needs to be sent by $\mathcal{DEV}$ to enable reconstructing the hash values at $\mathcal{MOD}$ side. The attestation report is assembled of a bitmap of executed control-flow edges and their iteration numbers as well as a single MSH measurement of the executed edges. The resulting attestation report is then sent to $\mathcal{MOD}$. $\mathcal{MOD}$ uses the bitmap to identify the executed edges and the number of times each edge was executed to identify benign behavior by the expected control-flow edge iterations. Therefore, it can detect more sophisticated attacks which often involve a loop iteration counter getting maliciously compromised, thus causing the loop to execute for an abnormally different number of times than expected. $\mathcal{MOD}$ is assumed to be more computationally resourceful than $\mathcal{DEV}$ and provisioned with more complex fine-grained policies and heuristics in order to detect more sophisticated non-control-data attacks (**A4** attacks). The activated paths for this mode are A, B, C, D, F, H, I, J, K, L, and M through the activated modules $\mathbf{M}_{1b}$, $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$, $\mathbf{M}_6$, $\mathbf{M}_7$, $\mathbf{M}_8$, $\mathbf{M}_9$, and $\mathbf{M}_{10}$.

C4. This configuration mode assumes a much stronger adversary capable of manipulating the actual instructions that are executed at runtime without compromising control flow. To detect such an attack, every instruction executed, besides control flow, is captured by both $\mathbf{M}_{1a}$ and $\mathbf{M}_{1b}$ respectively, and included into the cryptographic hash computations. As in **C3** mode, a final MSH measurement is sent along with a bitmap of executed control-flow edges and their iteration numbers to $\mathcal{MOD}$ in order to detect attacks **A3**, **A4** and **A5**. The activated paths for this mode are (A, B, C, D, F, H, I, J, K, L, and M) through the activated modules $\mathbf{M}_{1a}$, $\mathbf{M}_{1b}$, $\mathbf{M}_2$, $\mathbf{M}_3$, $\mathbf{M}_4$, $\mathbf{M}_6$, $\mathbf{M}_7$, $\mathbf{M}_8$, $\mathbf{M}_9$, and $\mathbf{M}_{10}$.

While providing progressively stronger and more sophisticated detection guarantees, configuration modes **C3** and **C4** come with significant latency that includes network communication with the remote moderator. On the other hand, modes **C1** and **C2** are performed on-device, thus incurring minimal (if any) clock cycle-level latency for detection/enforcement.

IV. CHASE: Hardware Architecture

We describe next the functionality of the hardware modules of CHASE used to enable the different configuration modes.

`Execution Tracer` $\mathbf{M}_1$ is tightly coupled to the processor pipeline. It tracks the execution flow of a software module/program by extracting several signals relevant to the execution from the processor pipeline at every clock cycle. It consists of two sub-modules: the Instruction Tracer which captures every instruction executed when mode **C4** is activated, and the Control-Flow Tracer which captures all signals relevant to control-flow transfers, as shown in Figure 2. This module requires tight interfacing with the Fetch and Decode stages of the pipeline to extract the program counter (PC), the instruction itself, and whether it is an indirect branch. Only indirect branches are exploitable by runtime attacks, and thus only these are captured and their destination addresses are extracted from the Decode stage.

`Bounds Checker` $\mathbf{M}_2$ controls which control-flow events are tracked and attested when either mode **C3** or **C4** is activated. For modes **C3** or **C4** only software modules that contribute to the generation of the requested data/message are attested during their execution (§ III-A). Therefore, the `Bounds Checker` compares the current PC with the address bounds of the relevant software modules. This requires two clock cycles and is interleaved with the operation of other CHASE modules for minimal performance overhead.

`Policy Access Mapper` $\mathbf{M}_3$ receives an address pointer to the policy memory from the `Execution Tracer` and maps it to corresponding cache entry in the `Policy Cache`, where the requested policy would be available. If the policy is already cached then the `Policy Access Mapper` fetches it from the `Policy Cache`. Otherwise, it issues a cache miss. The `Policy Access Mapper` also enables successive accesses to the `Policy Cache` in case more than one cache entry is required to store the policies for the pertinent control-flow transfer. The policies consist of the allowed source–destination address tuples and are organized in the policy memory per control-flow transfer, i.e., by source address, to achieve spatial locality in the `Policy Cache`, thus lower miss rates. As explained in § III-A, the policy memory addresses are instrumented within the program binary immediately after their corresponding indirect branch instructions, such that they get fetched anyway by the processor then invalidated (incurring no additional overhead). The `Execution Tracer` extracts the instrumented policy memory address before it gets invalidated.

`Policy Cache` $\mathbf{M}_4$ is accessed to fetch the cache line with the requested policy once the policy memory address is resolved by the `Policy Access Mapper`. The policy is then forwarded to either the `Control-Flow Verifier` or `Control-Flow Enforcer` depending on whether **C1** or **C2** is enabled respectively.

`Control-Flow Enforcer` $\mathbf{M}_5$ is tightly coupled with the processor Decode stage. For function calls and forward edges, the `Control-Flow Enforcer` ensures that the computed destination address matches one of the possible destination addresses in the relevant policy. For function returns or backward edges, the `Control-Flow Enforcer` utilizes the call-return matching stack to enforce returning to one of the call sites on the stack. Otherwise, it issues an interrupt signal to flush the pipeline and jump to an interrupt routine, which requires invasive integration with the pipeline.

`Control-Flow Verifier` $\mathbf{M}_6$ checks whether the computed control-flow destination address adheres to the pertinent policy by comparing it with the allowed destination addresses in parallel. If no match is found, then a control-flow violation is detected and communicated to $\mathcal{MOD}$. Execution may be gracefully interrupted and redirected to an application-dependent safe state. In case **C3** or **C4** is also enabled, the executed control-flow event is recorded in the metadata and included in the hash computation to report to $\mathcal{MOD}$. $\mathcal{MOD}$ can analyze the detected violation to confirm that it is indeed a violation and not a false positive. Otherwise, it can update the provisioned policies if necessary. The `Control-Flow Verifier` also informs the `Control-Flow (CF) Event Counter` with the executed control-flow event for further processing.

`Control-Flow (CF) Event Counter` $\mathbf{M}_7$ is only enabled for modes **C3** and **C4** and receives information from the `Control-Flow Verifier` on the executed control-flow transfers. The `CF Event Counter` generates and maintains a bitmap per software module that indicates which indirect edges and how many times they are executed by means of maintaining edge counters. It also maintains a list of control-flow transfers that were flagged as violating. At the end of the attestation epoch of **C3** or **C4**, the `CF Event Counter` outputs the metadata to be sent to $\mathcal{MOD}$ namely, the bitmaps, violated control-flow events and edge-counters.

`Hash Regulator`, **`Hash Engine`** and **`MSH Operator`** are only enabled in the two modes **C3** and **C4**. **`Hash Regulator`** $\mathbf{M}_8$ regulates the operation of the `Hash Engine` for different configuration modes and instructs it when to initialize/finalize a hash computation. When **C3** is enabled, it instructs the `Hash Engine` to compute a hash measurement over the executed control-flow event. It also guarantees that an executed control-flow transfer is hashed only once when it is encountered for the first time, thus avoiding exhausting the `Hash Engine` (with respect to power computation and clock cycles) in repeatedly computing the same hash values. When **C4** is enabled, it also forwards the executed instructions between consecutive control-flow events to the `Hash Engine` for hashing. Finally, it also regulates the forwarding of the computed hash values from the `Hash Engine` to the `MSH Operator`.

`Hash Engine` $\mathbf{M}_9$ is required to be a high-throughput and collision-resistant cryptographic hash function (**R5** in § II-B). We deploy BLAKE2 [4] as the underlying hash function for the computation of the multi-set hash value [14] that will be reported to $\mathcal{MOD}$. More details specific to our BLAKE2 instantiation are presented in § V.

`MSH Operator` $\mathbf{M}_{10}$ receives the computed hash values from the `Hash Engine` to continuously generate the intermediate and final multi-set hash (MSH) values. The `MSH Operator` performs additive multi-set hashing using simple arithmetic operations ($+$ and *mod*) [14]. At the end of the attestation epoch, the final MSH-hash value is sent along with metadata to $\mathcal{MOD}$ for further inspection. The execution of the `Hash Engine` and `MSH Operator` are interleaved to achieve minimal performance overhead.

V. Implementation and Evaluation

We prototype CHASE by extending the open-source RISC-V Pulpino [5] on a Zedboard Zynq evaluation board. Hardware modules were implemented in Verilog and integrated with the processor, along with integrating modifications to the processor pipeline for capturing the required execution signals and enabling control-flow enforcement. In the following, we highlight the most crucial details relevant to our proof-of-concept (PoC) implementation and evaluation.

A. Hardware Prototyping

`Policy Access Mapper & Policy Cache.` In our PoC, we deploy a direct-mapped `Policy Cache` and a simple *mod* function for the `Policy Access Mapper` that requires a single clock cycle. The policy is fetched from the mapped cache entry in the `Policy Cache`. Similar to conventional processor caches, the cache organization, mapping and replacement policies are design decisions and can be applied differently for the `Policy Cache`. Furthermore, cache misses in the `Policy Cache` would correspond to misses in the instruction cache. In case several cache lines need to be accessed for fetching the policies of a control-flow transfer, successive accesses to the `Policy Cache` are pipelined with the operation of the `Control-Flow Verifier` and `Control-Flow Enforcer` to maintain minimal performance overhead. In our PoC, we assume a 64KB cache and a 64B cache line, such that a tuple of 16 32-bit addresses (thus up to 16 policies for one source address) correspond to one cache line. We synthesize the cache using 8 instances of 64Kb Block RAMs (BRAMs), such that the complete cache line (storing 16 policies) is fetched in one cycle.

`Hash Engine & Hash Regulator.` Blake2 is deployed as the underlying cryptographic hash function for the computation of the multi-set hash [14]. We utilize the Blake2s version, which consists of 10 rounds of computation. In each round, the compression function is applied in parallel to the columns of the internal state and then in parallel to its diagonals. We build our `Hash Engine` implementation on top of the open-source Verilog implementation of Blake2s [6]. The `Hash Regulator` is a simple controller that regulates the operation of `Hash Engine` for different configuration modes and instructs it when to initialize or finalize a hash computation. In **C4**, instructions executed after a destination address of a control-flow transfer and the source address of the next control-flow transfer are split into blocks of 512 bits such that the blocks are hashed individually by `Hash Engine`. These values are forwarded continuously to the `MSH Operator` for the final MSH-hash value computations.

[4]https://blake2.net/
[5]https://github.com/pulp-platform/pulpino
[6]https://github.com/secworks/blake2s

B. Area & Performance Overhead

Area. In Table I, we show a breakdown of the area overhead of the CHASE modules. CHASE consumes $7,556$ lookup-tables (LUTs) and $5,040$ Registers/Flip Flops (FFs), approximately 50% of the baseline Pulpino logic (**R6** in § II-B). However, the `Hash Engine` incurs 50% of this overhead and requires at least 24 cycles to finalize a hash computation. The rounds of the `Hash Engine` implementation can be unrolled to achieve a significantly higher throughput (at the cost of increased area overhead).

Performance. We evaluate sample control-flow exploits (ROP and simple JOP attacks) with CHASE, and show that the detection latency of an illegal control-flow transfer is at least 3 clock cycles after an indirect branch is decoded. For instance, the total number of instructions in the Pixhawk[7] firmware, an open-source flight controller for drones (a timing-critical application), is $324,996$ instructions including $6,210$ indirect branches. For **C2**, this incurs a performance overhead of 6% on the application run time, assuming that all the policies are cached and only one cache line (maximum of 16 policies) is required per branch instruction. For **C1**, this latency does not affect the application run time (a requirement for such timing-critical applications), since the control-flow transfer is verified after its execution. However, this only holds as long as there are at least 3 clock cycles between consecutive indirect branch instructions, which is satisfied for this particular benchmark. For other cases where this is not true, only certain indirect branch instructions can be verified while others are discarded, or the binary can be re-instrumented accordingly. Nevertheless, the low detection latency guarantees that an impending exploit can be prevented in time (**R3**).

TABLE I: Breakdown of CHASE Area Overhead

	LUTs	FFs	Memory
`Execution Tracer`	210	128	1KB
`Bounds Checker`	630	$1,071$	4 KB
`Policy Access Mapper`	7	2	-
`Policy Cache`	-	4	64 KB
`Control-Flow Verifier`	230	140	-
`Control-Flow Enforcer`	1053	287	-
`Control-Flow Event Counter`	240	288	34 KB
`MSH Operator`	384	256	-
`Hash Regulator` & `Hash Engine`	$4,802$	$2,864$	-

VI. Security Analysis

CHASE aims to provide a configurable defense and cover the attack space described in § II-A. To achieve this, CHASE is required to provide *accurate*, *authentic* and *low-latency* enforcement or on-device verification of control-flow transfers for modes **C1** or **C2**. For **C3** and **C4**, CHASE is required to provide *accurate*, *complete*, *authentic*, and *fresh* attestation of control flow (as well as executed instructions in **C4**) of the program running on $\mathcal{DEV}$.

Attestation and Network Attacks. In modes **C3** and **C4**, the recorded control-flow events as well as the executed instructions (for **C4**) are measured into a compact cryptographically-secure additive multi-set hash (MSH) digest $\mathtt{MSH} = \big(\mathsf{hash}(r) + \Sigma\mathtt{Iter_i}.\mathsf{hash}(\mathtt{CFlow_i})\big) \bmod 2^{n}$, where r is a nonce, $\mathtt{CFlow_i}$ is an executed control-flow edge (source-destination address pair), $\mathtt{Iter_i}$ is the number of iterations of that edge, n is the bit length of $\mathtt{CFlow_i}$ and hash is the underlying hash function used (Blake2 in CHASE). In order to evade detection of control-flow/code manipulation attacks, $\mathcal{ADV}$ is required to find a sequence of control-flow events/instructions (another multi-set) that maps to the the same MSH

[7]http://pixhawk.org/

value. However, this is not feasible since the chosen additive MSH is multiset-collision resistant where the hardness of finding collisions is reduced to the hardness of breaking the underlying hash function, which is the second pre-image resistant Blake2 in our design (**R5**).

Moreover, every attestation report is authenticated by $\mathcal{DEV}$ along with a monotonic counter `ctr` using a cryptographically-secure digital signature $\sigma = \texttt{sign}\{sk_{\mathcal{DEV}}; \texttt{bitmap}_{\texttt{CF}} \| \texttt{Iter}_{\texttt{i}} \| r \| \texttt{MSH} \| \texttt{ctr}\}$ based on $\mathcal{DEV}$'s signing key $sk_{\mathcal{DEV}}$. $sk_{\mathcal{DEV}}$ is stored in hardware-protected memory that is only accessible by CHASE. The signature and secure storage of the key guarantee the *authenticity* of the report while the monotonic counter ensures its *freshness* (**R5**). Note that, the monotonic counter is backed in a non-volatile memory and is non-resettable even when $\mathcal{DEV}$ is reset. Finally, since attestation is coupled with program execution at $\mathcal{DEV}$, Time-of-Check-Time-of-Use (TOCTOU) attacks on attestation are prevented.

Malware and Code Injection Attacks. Recall that the adversary is incapable of modifying the software binary at load-time (malware injection) as well as the control-flow policies which are fetched from external memory into on-chip memory. This is due to static attestation which allows the detection of such tampering, thus mitigating attacks **A1**. Furthermore, code injection attacks are effectively prevented through DEP (W$\oplus$X)), thus mitigating attacks **A2**.

Runtime Code-Reuse Attacks. CHASE uses hardware modules that are tightly integrated with the processor to extract the control-flow information and executed instructions directly from the processor pipeline. This guarantees that all control-flow events and executed instructions are recorded. CHASE hardware also guarantees that all control-flow transfers are verified against their respective policies, thus, ensuring that the provisioned control-flow policies are accurately and completely enforced. The security guarantees with respect to the detection of runtime attacks are as good as the provisioned control-flow policies, and the code analysis that generated the policies. So long as any configuration modes in CHASE is enabled, at least every executed control-flow transfer is directly captured from the processor and either verified or enforced (for modes **C1** and **C2** respectively) using the provisioned control-flow policies or measured (for modes **C3** and **C4**). Thus, in order to ensure that no control-flow transfers are dropped, `Execution Tracer` of CHASE is padded with a First-In-First-Out (FIFO) structure that buffers incoming control-flow transfers. This is required for the unlikely event that multiple indirect jump instructions execute consecutively (**R1** and **R4**). While CHASE ensures that all buffered control-flow transfers are verified with their respective policies, the detection latency of potential violations is increased. Nevertheless, software developers are advised not to program multiple indirect branches consecutively.

This accurate tracking and enforcement/verification of control-flow edges in CHASE guarantees the detection of explicit control-flow hijacking attacks (**A3**). The call-return matching stack also provides additional guarantees on context-sensitive enforcement/verification of backward edges, i.e., returning to correct call sites. Moreover in **C3** and **C4**, tracking the number of times each edge iterates and sending the respective bitmap to $\mathcal{MOD}$ enables the detection of all data-oriented attacks that do not directly hijack the control flow but maliciously compromise the expected number of loop iterations (attacks **A4**). In **C4**, tracking and measuring every instruction executed (not only control flow) allows $\mathcal{MOD}$ to detect runtime code manipulation attacks (**A2** and **A5**). This assumes that $\mathcal{MOD}$ has knowledge of the program source code and the benign number of loop iterations for a given service/message by means of code analysis (see § II-A) (**R2**).

Finally, since CHASE is hardware-based, it cannot be compromised by malicious software. Moreover, all on-chip cache/memory utilized by CHASE, e.g., for policies, is hardware-protected and not mapped to software-accessible address space, and hence protected from remote software attacks.

Figure 3 shows an example of a control-flow violation at runtime. The solid arrows represent the expected benign execution path, while the dashed arrows respresent the violating control-flow edges, assuming the function pointer in the writable data memory was overwritten by the attacker. Assuming **C3** is enabled, the attestation report will differ from the reference in the bitmap of the executed edges, iterations per edge as well as the final MSH value computed.

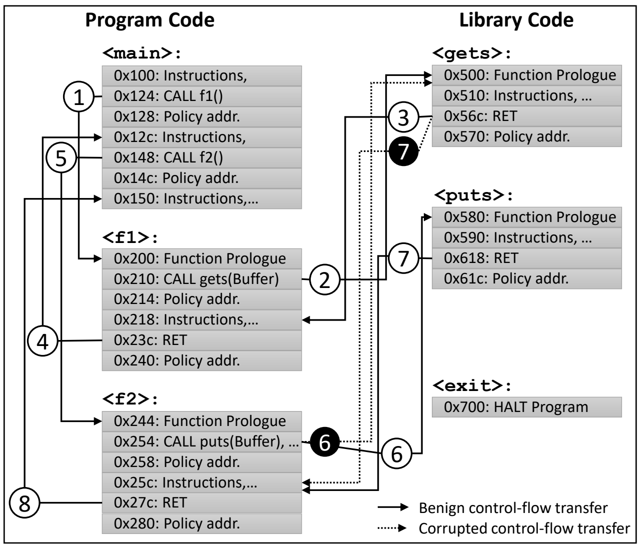


Fig. 3: Detection of a control-flow violation by CHASE. **Expected control-flow path (benign)**: ① (0x124, 0x200), ② (0x210, 0x500), ③ (0x56c, 0x218), ④ (0x23c, 0x12c), ⑤ (0x148, 0x244), ⑥ (0x254, 0x580), ⑦ (0x618, 0x25c), ⑧ (27c, 0x150), etc. **Traced control-flow path (corrupted)**: ① (0x124, 0x200), ② (0x210, 0x500), ③ (0x56c, 0x218), ④ (0x23c, 0x12c), ⑤ (0x148, 0x244), ❻ (0x254, 0x500), ❼ (0x56c, 0x25c), ⑧ (27c, 0x150), etc.

Physical Attacks. Expensive invasive/semi-invasive physical attacks are out of scope in this work, thus CHASE hardware is assumed secure against such attacks that would compromise its accuracy/functionality (**R4**). However, other realistic physical attacks that manipulate the program code at runtime, as well as fault injection attacks are captured by CHASE and detected by $\mathcal{MOD}$ in the mode **C4** since CHASE captures all executed instructions.

VII. Related Work

Static Attestation. Attestation aims at enabling a trusted third party to check the trustworthiness of the software on another device. Approaches to static attestation include: (1) Software-based attestation [15], [16] that allows the attestation of legacy and low-end computing devices while requiring no secure hardware but relying on strong assumptions, and thus have been attacked [17], (2) Hardware-based attestation [18], [19] that requires complex and/or security hardware (e.g., trusted platform module – TPM), and (3) Hybrid attestation [20]–[22] that is based on hardware/software co-design aiming at reducing the hardware security required for remote attestation. Static attestation, however, cannot detect runtime attacks.

Runtime Integrity. Many defenses have been proposed in recent years to mitigate runtime exploits [1]. Control-Flow Integrity

(CFI) [3] ensures that a program follows a valid path in its control-flow graph (CFG). However, CFI does not mitigate non-control-data and DOP attacks. Code randomization [23] randomizes the code layout, but a branch instruction can still be exploited to jump to the target address of choice. Code-Pointer Integrity (CPI) [24] aims at ensuring the integrity of code pointers but also does not mitigate non-control-data attacks. Defenses has been proposed for mitigating pure data-oriented attacks such as data-flow integrity enforcement/isolation [12], [25], [26], but they all incur a non-negligible performance overhead.

Runtime Attestation. Control-flow attestation was proposed in [4] and aimed to allow the verifier to attest the measure and record the control-flow path executed on the prover. However, code instrumentation was required and prohibitively high performance overhead was incurred on the prover. Subsequent schemes have been proposed [4], [9]–[11] to leverage hardware assistance for recording runtime execution events in parallel to program execution, and without code instrumentation, thus reducing the overhead on the prover significantly and tackling stronger adversarial capabilities [10], [11]. However, existing schemes each tackle different adversarial capabilities with no scheme providing a consolidated or configurable defense.

Defenses for Real-Time Systems. Some defenses have been recently proposed for providing integrity to real-time systems and the requirements of applying remote attestation to safety-critical systems was investigated recently in [27]. SeED [28] enables non-interactive attestation, while ERASMUS [29] proposes periodic self-measurements of the prover's software that are occasionally reported to a remote verifier, thus providing applicability to real-time systems, but only providing static integrity. DIAT [30] proposes data-integrity attestation for collaborating real-time systems but still incurs significant performance overhead to the application run time. ECFI [5] proposes a CFI mechanism for PLCs which gives priority to the PLC's runtime operation, yet it still incurs a performance overhead of up to 8.3%. Existing solutions cannot provide runtime security guarantees for a real-time system without incurring a performance overhead on the application, unlike CHASE.

VIII. Conclusion

In this work, we presented the first hardware-assisted security extension CHASE, that consolidates different modular defenses that can be configured at runtime to mitigate different adversarial capabilities, thus effectively covering a larger attack space than existing defenses. This enables the calibration of the security/performance trade-off by selecting the desired level of security and thus the corresponding performance overhead. Moreover, CHASE also provides a non-intrusive control-flow verification mechanism that does not affect the application run time, yet detects violations with minimal latency, making it applicable to timing-critical systems.

Acknowledgments. This work is supported by the German Research Foundation (DFG) within CRC 1119 CROSSING and HWSec, by the German Federal Ministry of Education and Research (BMBF) within CRISP, and by the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] L. Szekeres et al., "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (Oakland)*, 2013.

[2] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *ACM CCS*, 2007.

[3] M. Abadi et al., "Control-flow integrity: Principles, implementations, and applications," *ACM TISSEC*, vol. 13, no. 1, 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[4] T. Abera et al., "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[5] A. Abbasi et al., "ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers," in *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2017.

[6] K. Stouffer et al., "Guide to Industrial Control Systems (ICS) Security," *NIST special publication*, vol. 800, no. 82, pp. 16–16, 2011.

[7] L. Davi et al., "HAFIX: Hardware-Assisted Flow Integrity eXtension," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[8] D. Sullivan et al., "Strategy without Tactics: Policy-Agnostic Hardware-Enhanced Control-Flow Integrity," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016.

[9] G. Dessouky et al., "LO-FAT: Low-overhead control flow attestation in hardware," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2017.

[10] S. Zeitouni et al., "ATRIUM: Runtime Attestation Resilient Under Memory Attacks," in *International Conference On Computer Aided Design (ICCAD)*, 2017.

[11] G. Dessouky et al., "LiteHAX: Lightweight Hardware-assisted Attestation of Program Execution," in *International Conference On Computer Aided Design (ICCAD)*, 2018.

[12] C. Song et al., "HDFI: Hardware-Assisted Data-Flow Isolation," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[13] Hewlett-Packard, "Data execution prevention," 2006.

[14] D. Clarke et al., "Incremental multiset hash functions and their application to memory integrity checking," in *International conference on the theory and application of cryptology and information security*. Springer, 2003, pp. 188–207.

[15] R. Gardner, S. Garera, and A. Rubin, "Detecting Code Alteration by Creating a Temporary Memory Bottleneck," *IEEE Transactions on Information Forensics and Security*, 2009.

[16] Y. Li, J. M. McCune, and A. Perrig, "VIPER: Verifying the integrity of peripherals' firmware," in *ACM Conference on Computer and Communications Security*, 2011.

[17] G. Wurster, P. C. Van Oorschot, and A. Somayaji, "A Generic Attack on Checksumming-based Software Tamper Resistance," in *IEEE Symposium on Security and Privacy (Oakland)*, 2005.

[18] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot — A Coprocessor-based Kernel Runtime Integrity Monitor," in *USENIX Security Symposium*. USENIX Association, 2004.

[19] X. Kovah et al., "New Results for Timing-Based Attestation," in *IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[20] K. Eldefrawy et al., "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *Network and Distributed System Security Symposium*, 2012.

[21] P. Koeberl et al., "TrustLite: A Security Architecture for Tiny Embedded Devices," in *European Conference on Computer Systems*, 2014.

[22] A. Francillon et al., "A minimalist approach to remote attestation," in *Design, Automation & Test in Europe*, 2014.

[23] P. Larsen et al., "SoK: Automated software diversity," in *IEEE Symposium on Security and Privacy (Oakland)*, 2014.

[24] V. Kuznetsov et al., "Code-Pointer Integrity," in *USENIX Symposium on Operating Systems Design and Implementation (ODSI)*. USENIX Association, 2014.

[25] M. Castro et al., "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298470

[26] T. Nyman et al., "HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement," *CoRR*, vol. abs/1705.10295, 2017. [Online]. Available: http://arxiv.org/abs/1705.10295

[27] X. Carpent et al., "Invited: Reconciling Remote Attestation and Safety-Critical Operation on Simple IoT Devices," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2018.

[28] A. Ibrahim et al., "SeED: Secure Non-interactive Attestation for Embedded Devices," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2017.

[29] X. Carpent et al., "ERASMUS: Efficient Remote Attestation via Self-Measurement for Unattended Settings," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018.

[30] T. Abera et al., "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous System," in *Annual Network & Distributed System Security Symposium (NDSS)*, 2019.

HARDSCOPE: HARDENING EMBEDDED SYSTEMS AGAINST DATA-ORIENTED ATTACKS

[7] Thomas Nyman, Ghada Dessouky, Shaza Zeitouni, Aaro Lehikoinen, Andrew Paverd, N. Asokan, Ahmad-Reza Sadeghi. “HardScope: Hardening Embedded Systems Against Data-Oriented Attacks”. In Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC’19), 2019.

HardScope: Hardening Embedded Systems Against Data-Oriented Attacks

Thomas Nyman
Aalto University, Finland
thomas.nyman@aalto.fi

Ghada Dessouky
Technische Universität Darmstadt, Germany

Shaza Zeitouni
Technische Universität Darmstadt, Germany
{ghada.dessouky,shaza.zeitouni}@trust.tu-darmstadt.de

Aaro Lehikoinen
Aalto University, Finland
aaro.j.lehikoinen@aalto.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

N. Asokan
Aalto University, Finland
asokan@acm.org

Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

ABSTRACT

Memory-unsafe programming languages like C and C++ leave many (embedded) systems vulnerable to attacks like control-flow hijacking. However, defenses against control-flow attacks, such as (fine-grained) randomization or control-flow integrity are ineffective against data-oriented attacks and more expressive *Data-oriented Programming* (DOP) attacks that bypass state-of-the-art defenses.

We propose *run-time scope enforcement* (RSE), a novel approach that efficiently mitigates *all currently known DOP attacks* by enforcing compile-time memory safety constraints like variable visibility rules at run-time. We present Hardscope, a proof-of-concept implementation of hardware-assisted RSE for RISC-V, and show it has a low performance overhead of 3.2% for embedded benchmarks.

1 INTRODUCTION

Data-oriented attacks can influence program behavior without the need to modify control-flow data. Instead, they corrupt variables used by the program's decision making, or leak sensitive information from program memory. Such attacks are called non-control-data attacks [7]. Non-control-data attacks have been shown to allow attackers to forge user credentials, change security critical configuration parameters, bypass security checks, and escalate privileges. Recent work shows that it is even possible to generalize data-oriented attacks to construct full-blown malicious attacks with Turing-complete expressiveness, called *Data-Oriented Programming* (DOP) [15]. Such attacks are executed by carefully corrupting only non-control data over time to chain together sequences of operations on attacker-controlled input. DOP provides similar capabilities to attackers as return-oriented programming [26], but without breaking the victim program's control-flow integrity. This, combined with the ability for DOP to reuse virtually any data, makes preventing DOP attacks a significant and open challenge.

An extended version of the work available [22].

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317836

Existing defenses against control-flow attacks cannot prevent data-oriented attacks. Some defenses against non-control-data attacks (e.g., [5, 24]) protect individual pieces of (security-critical) data. Hu et al. [15] discuss various existing schemes that could reduce the number of DOP attacks, including memory safety, data-flow integrity, fine-grained data-plane randomization, and hardware/software fault isolation. However, they explain that existing approaches are either too coarse grained, or result in prohibitively high performance overheads. Without viable alternatives, and because effective defenses against control-flow attacks are already being deployed, DOP is likely to become the next appealing attack technique for run-time exploitation.

Goals and Contributions. We propose a new *efficient* defense against data-oriented attacks that effectively prevents *all currently known* DOP attacks. It can also be configured to prevent control-flow hijacking. The intuition behind our approach is simple: In *block structured languages* every variable has a *lexical scope*, denoting the block(s) of source code in which the variable is visible. All correct compilers enforce *variable scope* at compile-time by checking these variable visibility rules. All currently known DOP attacks, and many data-oriented attacks in general, violate variable scope rules at run-time, since there is no equivalent enforcement. Consequently, mechanisms for variable scope enforcement *at run-time* can significantly reduce the exposure to data-oriented attacks.

In this paper, we define the notion of *Run-time Scope Enforcement* (RSE) that provides *fine-grained compartmentalization* of data memory within programs. We then describe HardScope, a *hardware-assisted* RSE scheme. HardScope differs from existing defenses in the following important ways: a) it provides *complete meditation* of all variables accesses, b) it is *efficient*, incurring only a small performance overhead for embedded benchmarks, and c) it enables *context-specific* policies. This means that the same piece of code can be granted access to different memory locations depending on the context in which the code is executed. Our main contributions are:

- *Run-time Scope Enforcement*: A novel approach for fine-grained **context-specific memory isolation** within programs (Section 3) to defeat data-oriented attacks.

- *HardScope*: An open-source proof-of-concept implementation of hardware-assisted RSE on the RISC-V architecture that demonstrates **efficient memory compartmentalization** (Section 4).
- *Compiler support and APIs*: Compiler support for **protecting static and automatic variables at run-time** (Section 4.3) without requiring any developer input, and a **programmer's API** (Section 4.4) that allows developers to annotate dynamic allocations to complement the automated instrumentation.
- *Evaluation*: Analysis of RSE security guarantees (Section 5.1), and evaluation of HardScope's hardware area overhead and minimal performance impact (Section 5.2).

2 ADVERSARY MODEL & CHALLENGES

Adversary Model. We consider a powerful adversary who has full control over the data memory of the target program. This models buffer overflows and other memory corruption vulnerabilities (e.g., an externally controlled format string[1]) that could corrupt any data memory. However, the adversary cannot modify program code (W⊕X protection). Our adversary model is standard for run-time attacks and consistent with Hu et al.'s DOP attacks [15].

Challenges. Our goal is to prevent the above adversary from mounting DOP attacks. Since DOP attacks (similar to many other data-oriented attacks) require the adversary to modify and access data in unintended ways at run-time, these attacks can be prevented by a *run-time enforcement mechanism* that prevents any data access that would not be permitted during a compile-time check by a correct compiler. Designing a solution to meet this goal requires addressing the following significant challenges:

$\boxed{\text{C1}}$ *Run-time enforcement:* enforcing variable scopes at run-time requires information which is usually only available at compile-time.

$\boxed{\text{C2}}$ *Multi-granularity enforcement:* the enforcement mechanism must be configurable for any granularity of protection domain (subject) and protected region (object).

$\boxed{\text{C3}}$ *Context-specific enforcement:* enforcing different permissions on each invocation of the same subject (e.g., each function), to minimize the attack surface following the principle of *least privilege*.

$\boxed{\text{C4}}$ *Complete mediation:* protection domains cannot be allowed to increase their permissions accidentally or maliciously, and all memory accesses must be checked with only minimal performance impact and memory overhead.

3 DESIGN OVERVIEW

The high-level idea of HardScope is to extend the compiler to emit compile-time information about the visibility of variables, and to extend the underlying hardware to use this compiler-supplied information to dynamically create and update a set of memory access rules against which all memory accesses are checked.

Run-time enforcement. Machine code produced from languages such as C and C++ does not include information available to the compiler about variables and code blocks ($\boxed{\text{C1}}$). RSE needs this information to assign in-memory variables to specific *execution contexts*. To bridge this gap between compile-time lexical scope and run-time execution context, we modified the compiler to instrument the program code with special instructions that record which variables may be used by each code block. HardScope introduces an instruction set extension for this purpose (Section 4).

The compile-time components and behavior of HardScope are illustrated in Figure 1. An unmodified source code program (❶) is fed to the compiler (❷), which checks (as usual) that all variable accesses are correctly scoped. Our new *RSE Plug-in* (❸) in the compiler adds HardScope instructions (❹) at particular locations in the binary (e.g., at the start of functions). This results in a fully-functional program binary, instrumented with HardScope instructions that the HardScope hardware uses to create a set of rules against which all memory accesses can be checked at run-time.

Multi-granularity enforcement. We chose function-level compartmentalization as the granularity of isolation, since this is sufficient to mitigate all currently known DOP attacks (Section 5.1). However, RSE can also be implemented at other granularities (Section 4), without changes to the new HardScope hardware ($\boxed{\text{C2}}$).

Context-specific enforcement. Consider the program (❶) in Figure 1: function `C` receives two pointers and copies data from the first pointer to the second. It can be called from either function `A` or function `B` (call graph shown in Figure 2). In benign execution, variables x and y are only used in a *privileged* execution path, where access control checks prevent misuse (e.g., x could be a secret key). Function `B` contains an exploitable vulnerability allowing the attacker to control the pointers passed to function `C`. Since function `C` can be used to copy arbitrary data between two attacker-controlled pointers, this constitutes a DOP gadget. The attacker could use this to bypass the access control checks on variables `x` and `y` by accessing them through the unprivileged execution path.

HardScope prevents this by providing context-specific enforcement, in which different memory access rules can be associated with *each active instance* of a function ($\boxed{\text{C3}}$). To achieve this, the HardScope hardware creates memory access rules dynamically for each individual function invocation, and stores these in a data structure called the *Storage Region Stack* (SRS). The SRS is kept in hardware-isolated *protected memory*; only HardScope instructions can add or remove SRS entries. Each SRS entry defines an area of memory (e.g., the location of a variable) that may be accessed. The SRS is organized into *frames*; each frame contains all the entries for a particular execution context. The topmost SRS frame corresponds to the active execution context. On each memory access, e.g., load or store, HardScope validates that the memory address matches an entry in the topmost SRS frame.

Specifically, HardScope prevents the attack in Figure 2 as follows: The SRS for function `A` (❺) includes variables x and y, and the SRS for function `B` (❻) includes variables i and j (Figure 2). To allow function `C` to access certain variables, the calling function must use a special instruction (Figure 1 ❼) to *delegate* access to a variable to function `C`: e.g., function `A` must delegate access to x and y. For valid delegation, the calling function must already have access to the delegated variables. Even though the attacker can still manipulate the pointers in function `B`, this function does not have access to x and y (no corresponding SRS entries) and hence it cannot delegate access to these variables to function `C`.

[1] CWE-134: Use of Externally-Controlled Format String
https://cwe.mitre.org/data/definitions/134.html

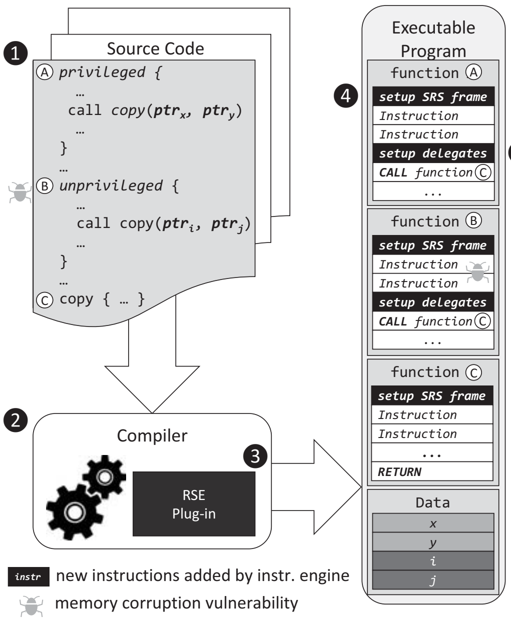
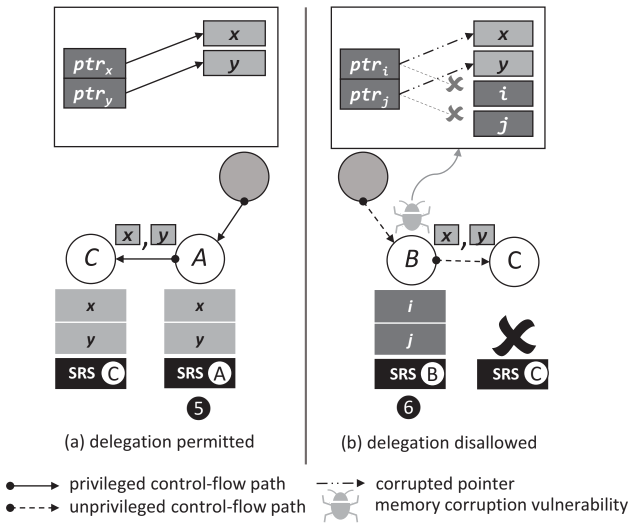


Figure 1: Compile-phase design of HardScope. Run-time memory accesses via pointers ptr_x, ptr_y are limited to variables x and y, while ptr_i, ptr_j are limited to i and j.



ptr_x
ptr_y
x
y
ptr_i
ptr_j
i
j
C
A
B
SRS C
SRS A
SRS B
5
6
(a) delegation permitted
(b) delegation disallowed
privileged control-flow path
unprivileged control-flow path
corrupted pointer
memory corruption vulnerability

Figure 2: Run-time design of HardScope showing the call graph of program in Figure 1. In (a), access to variables x and y is successfully delegated from A to C. In (b), function B should not have access to x and y, but a memory corruption vulnerability in B is used to corrupt ptr_i and ptr_j to point to x and y instead of i and j. HardScope prevents B from accessing or delegating x and y.

4 IMPLEMENTATION

We developed a proof-of-concept hardware implementation of HardScope and integrated it into the open-source RISC-V Pulpino core.[2] HardScope extends the RISC-V instruction set with seven new SRS management instructions, as shown in Table 1. We augmented the GCC compiler to incorporate a proof-of-concept RSE plug-in and a modified RISC-V backend to automatically instrument C programs with the relevant HardScope instructions. These protect static and automatic variables at run-time without requiring any changes to program code. We also developed a HardScope **Programmer's API** (Section 4.4) that allows developers to annotate dynamic allocations to complement the automated instrumentation. HardScope itself is architecture-agnostic; our choice of RISC-V and Pulpino is due to the open-source nature of the ISA and the RTL implementation, thus enabling us to prototype our solution.

4.1 Instructions

The `sbent` and `sbxit` instructions are used to mark the beginning and end of each execution context. HardScope uses these instructions to track when HardScope is first enabled and when the execution context changes, and thus when new enforcement rules should be loaded in the SRS. `sbent` pushes a new frame on top of the SRS, whilst `sbxit` pops the topmost SRS frame. Program execution starts with an empty SRS and HardScope enforcement is initially disabled. HardScope is enabled by the first `sbent`, and remains enabled until a matching `sbxit` empties the stack.

The `sradd` and `srdda` instructions create an SRS entry in the current (topmost) SRS frame. HardScope uses these instructions to determine the bounds of memory areas that the current execution context is allowed to access. The two operands set the *base* and *limit* address of the storage region respectively. An optional offset is added to to either the limit (`sradd`) or base (`srdda`) register operand.

The `srdel` instruction removes the specified number of SRS entries from the current SRS frame (last in first out). It allows the program to drop unneeded memory access privileges without changing execution context.

The `srdlg` and `srdsub` instructions delegate an SRS entry from the currently executing function either to an invoked callee function or to the caller when the current function returns. HardScope uses these instructions to derive SRS entries for data flows which are not known at compile-time, such as context-specific accesses (Section 3). The operands specify an address to determine which memory address to delegate. The resulting memory address is compared with the current SRS entries and if a match is found, the most recent matching entry is copied to the *next execution context entered.* If the delegation is followed by a `sbent`, the delegated entry is added to the newly created SRS frame. If the delegation is followed by a `sbxit`, the delegated entry is added to the caller's SRS frame.

The `srdsub` instruction is used to delegate a new SRS entry that is a subset of an existing SRS entry. It takes the same operands as `sradd`. If the new subdivided memory region is a subset of an existing SRS entry in the current SRS frame, a new SRS entry is created for a *sub-region* using the new base and limit.

If no matching entry is found in the SRS when `srdlg` or `srdsub` execute, no entry is delegated. This prevents the use of `srdsub` to elevate the access rights of the next execution context beyond the rights of the current, but allows the delegation instructions to be applied to pointers which are not dereferenced directly in the current context. These include null-pointers and intentionally created out-of-scope pointers (e.g., via the use of pointer arithmetic)

[2] http://www.pulp-platform.org/

Table 1: HardScope Instructions. *Operands* lists valid combinations of operands: `r`*n* is a register, `imm` is an immediate value, and `imm(r`*n*`)` is a register to which an immediate offset is added. *Cycles* indicates the number of cycles consumed at execute stage.

Mnemonic	Name	Operands	Cycles
`sbent`	**s**cope **b**lock **ent**er	n/a	1 (+ *N*)
`sbxit`	**s**cope **b**lock e**xit**	n/a	1 (+ *N*)
`sradd`	**s**torage **r**egion **add**	`r1, imm(r2)`	1
`srdda`	**s**torage **r**egion **dda** (reverse add)	`imm(r1), r2`	1
`srdel`	**s**torage **r**egion **del**ete	`imm(r1)` `imm`	1 (+ 1)
`srdlg`	**s**torage **r**egion **del**e**g**ate	`imm(r1)` `imm`	1 (+ 1)
`srdsub`	**s**torage **r**egion **d**elegate **sub**-region	`r1, imm(r2)`	1 (+ 1)

that are passed to callees for which they are in scope (e.g., accessor functions that receive opaque pointers as arguments).

4.2 Hardware Implementation

We modified the instruction decoding stage of the processor pipeline to interpret the new instructions (Section 4.1). After decoding, the appropriate control signals are sent to the HardScope unit, which realizes the execute stage of the new instructions. Figure 3 shows the main components of the HardScope unit: the *SRS controller* (❶), dedicated memory to hold the SRS (❷), and three register banks (❸, ❹, ❺). The *active bank* (❸) holds the entries in the SRS frame for the current execution context enabling each memory access to be compared against *all active entries* efficiently. The *spare bank* (❹) holds entries delegated via `srdlg` and `srdsub` before a HardScope context switch occurs. It allows delegated entries for the *next execution context* to be accumulated ahead of time. When a HardScope context switch occurs, the spare bank becomes the active bank (and vice versa), thus activating the delegated entries. The third bank (❺) is used as a cache to hold a copy of the topmost frame of the SRS. This reduces the latency when the topmost SRS frame is transferred between the stack memory and the spare bank.

When executing `sbent`, the controller activates the spare bank and transfers the contents of the currently active bank to the cache (❻) in a single cycle. The bank that held the previously active frame becomes the spare, and can be used for subsequent delegations. The entries in the cache must be stored for future use, and are transferred to the SRS in protected memory (❼) over at most *N* subsequent cycles, where *N* is the maximum number of entries in the cache. During this time, the CPU continues to execute subsequent instructions normally until a new HardScope context switch occurs. Only if a HardScope context switch occurs before the cache has been emptied does the processor stall until the transfer is complete.

When executing `sbxit`, the controller copies the SRS frame from the cache into the spare bank (❽) while retaining delegated entries (i.e., activating the entries that are already in the spare bank). The SRS frame in the previously active bank is no longer needed and is discarded. This executes in a single cycle. The cache, which now holds an out-of-date copy of the active frame, is updated with the topmost SRS frame from the protected memory (❾), which takes at most *N* cycles, where *N* is the number of entries in the topmost SRS frame in memory. This does not stall the processor unless another `sbxit` is encountered before the cache is fully populated, in which case the CPU stalls until the next frame is available. However, if

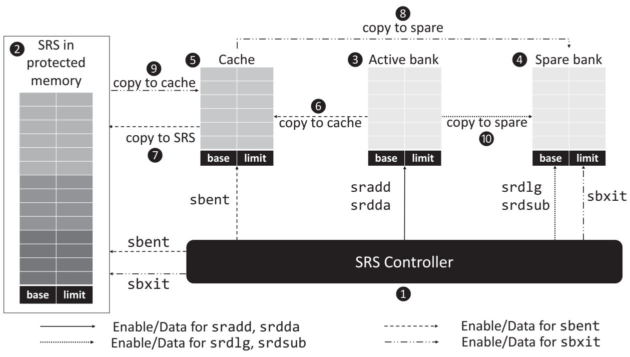


Figure 3: HardScope hardware architecture.

an `sbent` is encountered before the cache is fully populated, the partial cache is discarded and replaced with the contents of the active bank, without stalling.

The `sradd` and `srdda` instructions always operate on the active bank. When executing `srdsub`, the controller checks the active bank for an entry containing the given memory region and, if found, adds the new sub-entry to the spare bank. Similarly, in `srdlg`, the controller checks for the matching entry in the active bank and, if found, copies the entry to the spare bank (❿). The `srdlg` and `srdsub` instructions require an additional cycle only if followed immediately by a `sbent` or `sbxit`.

Integrating HardScope into the processor pipeline also required modifying the memory access stage to intercept all memory access requests to the load/store unit. At each `load` or `store` instruction, the requested memory address and the number of requested bytes (one byte, half-word (two bytes), or word (four bytes)) are intercepted and forwarded to the SRS controller, which compares it against all entries in the active bank. The registers in each bank are wired to comparators such that all entries in the bank are checked *in parallel*. If a match is found, i.e. the requested address range is a subset of any of the active entries, then the memory access is granted by the processor's load/store unit, otherwise a hardware fault exception is raised. We design and integrate HardScope to the processor pipeline such that no additional clock cycle latency is incurred to the baseline `load` and `store` instructions.

4.3 Software Instrumentation

Our RSE GCC plug-in and the modified RISC-V backend currently supports automatic instrumentation of C programs at *function granularity* to protect the 1) call stack frame including local variables, return address and other return state information, 2) arguments passed on the stack, 3) heap objects, and 4) global and static variables. The beginning of each distinct execution context is marked by inserting a single `sbent` instruction at the function call site just before the jump instruction. The end of an execution context is marked by inserting an `sbxit` instruction just before the return in the callee function. In Section 5 we show that function-level isolation is sufficient to mitigate all currently known DOP attacks. However, RSE can also be implemented at other granularities, without changes to the HardScope instructions, by inserting `sbent` and `sbxit` instructions around the instructions that comprise a distinct execution context.

4.4 HardScope Programmer's API

Deeply Nested Pointers. The HardScope Programmer's API enables the handling of code that uses deeply nested pointers e.g., traversing linked lists. This type of code is a challenge for automated instrumentation because e.g., passing the head of a linked list to a function that iterates through the list would require delegation of an SRS entry for each element of the list. Since the number of SRS entries (per frame) is constrained by the HardScope hardware (see Section 4.2), this leads to suboptimal use of HardScope hardware resources and an increased cost in HardScope context switches due to more frequent stalls at run-time. Instead, we propose a programming pattern using the HardScope Programmer's API where one `sradd` instruction is added before the dereference of member pointers to linked member elements, and one `srdel` is added after the dereference. This enables effective yet secure traversal of linked lists and other data structures containing nested pointers.

Heap object allocation. We implemented a wrapper on top of the C standard library `malloc()` function that creates SRS entries for heap allocations, and delegates these to the caller. Other library functions can be similarly wrapped to allow HardScope-instrumented code to be linked against uninstrumented libraries.

5 EVALUATION

HardScope meets the stated challenges (Section 2) as follows:

$\boxed{\text{C1}}$ **Run-time enforcement.** The RSE GCC Plug-in infers and emits the necessary HardScope instructions to manage the SRS for stack and global data, as well as dynamic allocations that follow a well-defined pattern. The HardScope Programmer's API allows handling code that is not automatically instrumentable, e.g., uses deeply nested pointers.

$\boxed{\text{C2}}$ **Multi-granularity enforcement.** HardScope can enforce policies with either coarser or finer granularity of execution contexts with the appropriate instrumentation ($\boxed{\text{C2}}$). For instance, HardScope can isolate the function prologue and epilogue from the function body, and protect return addresses on the stack from memory errors in the function body to prevent control-flow hijacking.

$\boxed{\text{C3}}$ **Context-specific enforcement.** In HardScope, the active SRS entries can differ between different invocations of the same subject, depending on which entries have been delegated to this subject (e.g., variables passed to a function by its caller or callee).

$\boxed{\text{C4}}$ **Complete mediation.** HardScope hardware checks every memory access against the active set of SRS entries; accesses without matching entries will fail. Therefore only compiler-admissible memory accesses are allowed.

Instructions that create rules at run-time could potentially be used as *confused deputies*. In a *confused deputy attack*, the attacker attempts to subvert the RSE property by misusing existing HardScope instructions at run-time to create unintended rules. Our design ensures that no such instructions are available to the attacker. Rules for static allocations (stack and global variables) are encoded directly into the instructions. Since these cannot be modified at run-time, they cannot be used as confused deputies.

Instructions that create rules that are determined at run-time are found within memory allocators, e.g., `malloc()`, or code that deals with deeply nested pointers, e.g., iterators annotated using the HardScope Programmer's API. It is reasonable to assume that memory allocators are trusted (or at least that allocations are not influencable by the attacker). We recommend that manually annotated code is vetted for allocators that create rules at run-time. Furthermore, an attacker can only initiate a confused deputy attack if he already controls some part of the code, which is very difficult since every memory access in the instrumented program is checked by the HardScope hardware.

5.1 Security Evaluation

We replicated the DOP attack by Hu et al. [15] and ported the code to Pulpino to evaluate the effectiveness of HardScope. Although it was not possible to port the complete victim ProFTPD server to our FPGA testbed, we focussed on the vulnerable `sreplace()` function [15]. All enforcement rules in our experiments are derived *without any developer annotations* – the GCC intermediate representation contains all information necessary for compile-time instrumentation, including: stack-frame sizes, global variable accesses, function calls, parameters, and return values. Function-granularity isolation is sufficient to prevent the attack.

We verified experimentally *four* ways in which RSE prevents this DOP attack: 1) it prevents the initial memory violation in `sreplace()` as it enforces the indended bounds of input and output buffers when operated on by an unsafe string copy operation (`strncpy()` with incorrect buffer length), 2) it prevents the attack from keeping internal state in unused areas of the program's data section, 3) it denies access to global variables which are accessed by the attack out of their normal context, 4) it denies access to static variables which should only be accessible by code wihin the same compilation unit. We discuss each of these in detail in the extended version of this article [22]. Any one of these would be sufficient to stop the attack, and thus the existence of four distinct mitigations demonstrates the effectiveness of RSE's layered defense strategy.

5.2 Performance and Area Evaluation

Performance overhead. We ran CoreMark[3], a standard performance benchmark for embedded systems, with varying iteration counts on a HardScope-augmented Pulpino synthesized on a Xilinx Zynq-7020 ZedBoard. We observed an average overall performance overhead of 3.2% compared to the execution of unmodified CoreMark on the unmodified Pulpino SoC. All instrumentation in CoreMark was automatically generated by our extended GCC compiler resulting in the binary size increasing by 11%. The number of entries required per SRS frame varied throughout execution between 1 and 23. The overall maximum SRS size was 71 entries in 11 frames, resulting in a memory overhead of 573 bytes (64 bits per entry + 4 bits per frame to record the number of entries).

Area and memory utilization. The area utilization depends primarily on the size of active, spare and cache banks (i.e., the number of entries per frame). All three banks are mapped to logic to guarantee single-cycle access parallel checking of all frame entries. The area utilization increases linearly as the number of entries configured per frame increases (for a fixed number of frames), since more entries must be checked in parallel. For a protected memory size of 8 entries × 16 frames, HardScope utilizes 4, 572 LUTs, 1, 760 registers, and one 36 kB block RAM (RAMB36). For a 32 entries ×

[3]http://www.eembc.org/coremark/faq.php

16 frames configuration (required for the CoreMark performance evaluation above), HardScope utilizes 30, 520 LUTs, 6, 362 registers, and one 36 kB block RAM (RAMB36).

6 RELATED WORK

Various software-only and hardware-assisted memory safety technologies have been proposed and/or deployed (e.g., [2–6, 8, 11, 13, 17–19, 24, 25]). We discuss approaches that aim to mitigate data-oriented attacks in detail in the extended version of this article [22].

Software-only defenses (e.g. DFI [6] and SoftBound [20]) can offer strong security guarantees, but their usefulness is limited by high performance overhead, and by requiring changes to the system software architecture. Consequently, the granularity of enforcement of deployed defenses are often relaxed in favor of improved performance. Memory-safe dialects of C (e.g., CCured [21], Cyclone [16], and Checked C [12]) retrofit C with compile- and/or run-time checks that prevent memory errors from occuring. However, such dialects only benefit programs which are modified to make use of enhanced language features, also incur considerable run-time overhead [16, 21], or preclude certain C features [12].

Hardware-assisted defenses (e.g., BIMA [19], HDFI [27], and CHERI [28]) promise to drastically improve the performance overhead compared to software-based defenses. However, recent advances in attacks against bounds-checking approaches [14] suggest that low-fat pointer schemes which enforce allocation bounds rather than object bounds, such as BIMA [19] are exploitable. On the other hand approaches that track object bounds in separate storage, e.g., Intel MPX [23], HardBound [11], are not faster nor more memory effient than sofware-based approaches. Hardware-assisted tagged memory allow efficient enforcement of memoru access policies, but unlike HardScope only support a small number of simultaneour protection domains (e.g. two domains in HDFI [27]). CHERI [28] is a hardware-assisted capability model that can support various protection models, but requires program re-engineering.

Run-time attestation schemes [1, 9, 10, 29] can only detect, but not prevent, control-flow and non-control-data attacks.

Although HardScope shares many of the same goals as the above schemes, it differs in several fundamental aspects. Compared to software-based schemes (e.g., DFI [6] and SoftBound [20]), HardScope has significantly lower overhead, does not require whole-program static analysis, and can enforce context-specific policies for individual function invocations. HardScope RSE policies can be instantiated for a large class of programs without additional input from developers (cf., YARRA [24]), or software re-engineering (cf., CHERI). HardScope reduces the metadata needed at execution time to the rules for active execution contexts. Active rules are cached in on-chip memory, to enable access checks with no overhead.

7 CONCLUSION

By implementing and evaluating HardScope, we demonstrated that RSE is an effective approach to protect against data-oriented attacks. HardScope can also enforce memory isolation at coarser or finer granularity, to enable different memory protection strategies.

We provide 1) our enhanced GCC compiler; 2) instrumented binaries of our test programs; and 3) a RISC-V simulator with support for HardScope instructions at https://goo.gl/TAjLxy.

ACKNOWLEDGMENTS

This work was supported by the German Science Foundation CRC 1119 CROSSING projects S2 and P3, the German Federal Ministry of Education and Research (BMBF) within CRISP, the EU's Horizon 2020 research and innovation program under grant nr. 643964 (SUPERCLOUD), Business Finland under grant nr. 3881/31/2016 (CloSer), Academy of Finland under grant nr. 309994 (SELIoT), and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

REFERENCES

[1] Tigist Abera et al. 2016. C-FLAT: Control-Flow Attestation for Embedded Systems Software. In *Proc. ACM CCS '16*. 743–754.

[2] Periklis Akritidis et al. 2008. Preventing Memory Error Exploits with WIT. In *Proc. IEEE S&P '08*. 263–277.

[3] Sandeep Bhatkar and R. Sekar. 2008. Data Space Randomization. In *Proc. DIMWA '08*. 1–22.

[4] Cristian Cadar et al. 2008. *Data Randomization*. Technical Report MSR-TR-2008-120. Microsoft Research.

[5] Miguel Castro et al. 2009. Fast Byte-granularity Software Fault Isolation. In *Proc. ACM SOSP '09*. 45–58.

[6] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proc. USENIX OSDI '06*. 147–160.

[7] Shuo Chen et al. 2005. Non-control-data Attacks Are Realistic Threats. In *Proc. USENIX Security '05*. 12–12.

[8] Long Cheng, Ke Tian, and Danfeng (Daphne) Yao. 2017. Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks. In *Proc. ACM ACSAC '17*. 315–326.

[9] Ghada Dessouky et al. 2017. LO-FAT: Low-Overhead Control Flow ATtestation in Hardware. In *Proc. ACM/EDAC/IEEE DAC '17*. 24:1–24:6.

[10] Ghada Dessouky et al. 2018. LiteHAX: Lightweight Hardware-assisted Attestation of Program Execution. In *ICCAD '18*.

[11] Joe Devietti et al. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proc. ACM ASPLOS '08*. 103–114.

[12] Archibald Samuel Elliott et al. 2018. Checked C: Making C Safe by Extension. In *Proc. IEEE SecDev '18*. 53–60.

[13] Úlfar Erlingsson et al. 2006. XFI: Software Guards for System Address Spaces. In *Proc. USENIX OSDI '06*. 75–88.

[14] Ronald Gil, Hamed Okhravi, and Howard E. Shrobe. 2018. There's a Hole in the Bottom of the C: On the Effectiveness of Allocation Protection. In *Proc. IEEE SecDev '18*. 102–109.

[15] Hong Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proc. IEEE S&P '16*. 969–986.

[16] Trevor Jim et al. 2002. Cyclone: A Safe Dialect of C. In *Proc. USENIX ATC '02*. 275–288.

[17] Dmitrii Kuvaiskii et al. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proc. ACM EuroSys '17*. 205–221.

[18] Volodymyr Kuznetsov et al. 2014. Code-pointer Integrity. In *Proc. USENIX OSDI '14*. 147–163.

[19] Albert Kwon et al. 2013. Low-fat Pointers: Compact Encoding and Efficient Gate-level Implementation of Fat Pointers for Spatial Safety and Capability-based Security. In *Proc. ACM CCS '13*. 721–732.

[20] Santosh Nagarakatte et al. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proc. ACM PLDI '09*. 245–258.

[21] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proc. ACM POPL '02*. 128–139.

[22] Thomas Nyman et al. 2017. HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement. https://arxiv.org/abs/1705.10295

[23] Oleksii Oleksenko et al. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. https://arxiv.org/abs/1702.00719.

[24] C. Schlesinger et al. 2011. Modular Protections against Non-control Data Attacks. In *Proc. IEEE CSF '11*. 131–145.

[25] Konstantin Serebryany et al. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC '12*. 309–318.

[26] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. ACM CCS '07*. 552–561.

[27] C. Song et al. 2016. HDFI: Hardware-Assisted Data-Flow Isolation. In *Proc. IEEE S&P '16*. 1–17.

[28] Jonathan Woodruff et al. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proc. IEEE ISCA '14*. 457–468.

[29] Shaza Zeitouni et al. 2017. ATRIUM: Runtime Attestation Resilient Under Memory Attacks.. In *ICCAD '17*.

SOK: SECURE FPGA MULTI-TENANCY IN THE CLOUD: CHALLENGES AND OPPORTUNITIES

[8] Ghada Dessouky, Ahmad-Reza Sadeghi, Shaza Zeitouni. “SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities”. In Proceedings of the 6th IEEE European Symposium on Security and Privacy (EuroS&P’21), 2021.

SoK: Secure FPGA Multi-Tenancy in the Cloud: Challenges and Opportunities

Ghada Dessouky
Technische Universität Darmstadt
Darmstadt, Germany
ghada.dessouky@trust.tu-darmstadt.de

Ahmad-Reza Sadeghi
Technische Universität Darmstadt
Darmstadt, Germany
ahmad.sadeghi@trust.tu-darmstadt.de

Shaza Zeitouni
Technische Universität Darmstadt
Darmstadt, Germany
shaza.zeitouni@trust.tu-darmstadt.de

***Abstract*—Field Programmable Gate Arrays (FPGAs) are increasingly deployed in datacenters due to their inherent flexibility over ASICs or GPUs that makes them an ideal processing unit for emerging and dynamic area of deep learning and other techniques and algorithms that are rapidly evolving. To maximize their utilization in the cloud, researchers have proposed the spatial *multi-tenant* deployment model, where the FPGA fabric is simultaneously shared among mutually distrusting tenants. This is enabled by leveraging the partial reconfiguration capability of FPGAs.**

In this paper, we systematize the research work on multi-tenant FPGAs in cloud computing settings and highlight their adversary models, security guarantees, as well as their fundamental security and privacy related shortcomings. We further categorize existing research works that demonstrate a new class of remotely-exploitable physical attacks on multi-tenant FPGAs by malicious tenants sharing physical resources with the victims. Through investigating end-to-end multi-tenant FPGA deployment comprehensively, we reveal that these attacks represent only one dimension of the problem, while various open security and privacy challenges remain unaddressed. We conclude with our insights on future research challenges and open opportunities.

***Index Terms*—Cloud FPGA Security, FPGA Multi-tenancy, FPGA-based Acceleration, FPGA-based Trusted Computing**

1. Introduction

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be (re)programmed after fabrication, as opposed to Application-Specific Integrated Circuits (ASICs), in order to implement custom functionality in hardware. FPGAs provide more flexible computing fabric than their ASIC counterparts, yet higher throughput and more computing power than their software counterparts. They require lower energy consumption, and given their continuously increasing capacities, they have been perceived to bring the best of both hardware and software worlds. In their steady growth and penetration of different application domains, FPGAs have also made their way into hardware acceleration of machine learning applications among other compute-intensive services. More recently, FPGAs have been increasingly adopted in datacenters to accelerate cloud-based services, such as by Microsoft [1], [2]. Other enterprises, such as Amazon [3], are offering clients to rent FPGAs in the cloud which they can freely configure with their own logic.

Multi-tenant FPGAs. To maximize FPGA utilization and the return-on-investment in cloud computing, researchers have proposed to share a single FPGA fabric among multiple users [4]–[10] by leveraging the *partial reconfiguration* property of FPGAs. This key distinguishing feature of FPGAs allows to reconfigure the functionality of a part, or *region* of the FPGA while being deployed in-field (at runtime). The reconfiguration property of FPGAs, both normal and partial, has led to the notion of *virtualized* or *multi-tenant FPGAs*, which are used interchangeably to refer to FPGAs shared among several tenants in the cloud. While this distinction is often not clearly outlined, FPGA sharing/multiplexing/partitioning can occur either *temporally* or *spatially*. Temporal sharing is where the FPGA device fabric, or the accelerator configured thereon, is used by different tenants at different time slots. This is the more conventional FPGA sharing model typically deployed in industry solutions. More recently, however, researchers have been investigating spatial sharing, where multiple tenants' designs can be co-located on the same physical FPGA device simultaneously, while occupying different logically isolated FPGA regions. Although this approach is principally possible and would further boost utilization, such a setting is not yet deployed in cloud computing architectures. It introduces a new threat landscape which we systematically investigate in this work.

Remotely-Exploitable Physical Attacks. Recent academic works have demonstrated the feasibility of remotely-exploitable physical attacks in multi-tenant settings, with a particular focus on spatial multi-tenancy. Such attacks have been shown to compromise the availability (e.g., DoS attacks) [11], [12], or integrity (e.g., fault injection attacks) [13], [14], or confidentiality (e.g., side- or covert-channel attacks) [15]–[30] of the victim's logic on the FPGA. The root cause for most of these attacks is that both the victim and malicious tenants 1) have their logic, while logically isolated, still co-located on an underlying fabric that shares resources such as the power supply system and are thus not physically isolated, and 2) have the freedom to configure their allocated regions with any (malicious) hardware logic of their choice.

FPGA-Based Trusted Computing. Besides the focus on boosting performance and utilization of FPGA resources and the emerging attack surface, literature on FPGA-based accelerated computing covers other emerging industry trends such as datacenter disaggregation [31] and edge cloud computing [32]. Nevertheless, very little has been invested in the direction of FPGA-based trusted computing [33]–[35] that aims to protect clients'

Intellectual Property (IP) on cloud FPGAs. Remotely-exploitable physical attacks, while clearly a threat, are indeed not the only security challenge stemming from these deployment settings. Other fundamental questions remain: how is clients' IP protection assured? How can FPGAs and their toolchains enable a minimal Trusted Computing Base (TCB) in order to provide the required security guarantees for clients and cloud service providers in different deployment scenarios?

Contributions. In this work we aim to provide a comprehensive overview of the security concerns in multi-tenant cloud FPGAs and shed light on specific directions for future research to address these challenges. Our main contributions are as follows.

- Introduction to the state-of-the-art trends in multi-tenant FPGA-based cloud computing.
- Systematization of the various security requirements and challenges in FPGA-based cloud computing and potential approaches to address them.
- Systematization of the different attacks in FPGA-based cloud computing. We classify these attacks into two major categories. The first category is remotely-exploitable physical attacks (remote physical attacks for brevity) that stem from the configurable nature of FPGAs where clients can freely configure their (malicious) hardware circuits on the FPGA. We further classify these attacks depending on the deployment model assumed, either spatial or temporal multi-tenancy. The second category includes more classical attacks that do not require the configuration of malicious primitives on the FPGAs and have their counterparts in CPU- or GPU-based computing.
- Systematization of defenses based on the different security requirements for secure multi-tenant FPGA settings (spatial and temporal).
- Insights on the open challenges and opportunities in enabling FPGA-based *trusted computing* in the cloud.

2. Field Programmable Gate Arrays

FPGAs are integrated circuits that can be electrically programmed or *configured* by end users to implement different digital circuits. Compared to ASICs, FPGAs are cost-effective and have shorter time-to-market. Moreover, unlike ASICs, FPGAs can be reconfigured to overwrite or update an existing design. However, the flexible nature of FPGAs comes at additional costs in terms of area, power consumption and performance, which are mainly attributed to the programmable interconnect of FPGAs [36]. FPGAs consist of three major components: configurable logic elements, configurable interconnects and input/output (I/O) blocks, which provide off-chip connections. Configurable logic blocks are connected together and to I/O blocks through the programmable routing interconnects to form the desired functionality. Other components of FPGAs are memory blocks (BRAM) and digital signal processing blocks (DSPs) for high-performance DSP applications, as shown in Fig. 1b [36].

FPGAs are usually integrated on printed circuit boards (PCBs) Fig. 1a, such that a single board can have one or more FPGA chips along with other components such as external interfaces to a computing platform, e.g., PCIe, USB, etc. A single FPGA chip can pack one or multiple dies, by leveraging advances in 3D integration and packaging, as in recent FPGA chips, e.g., Intel Stratix 10 and Xilinx Virtex UltraScale+. We show this distinction, since academic proposals called for leveraging the different chips per PCB or different dies per chip in the multi-tenant FPGAs in the cloud.

Design flow refers to the steps required to convert an abstract circuit description, written in a hardware description language, such as Verilog or VHDL, into a functioning circuit on a target FPGA. In *synthesis*, an abstract form of circuit description (functional, behavioral or structural) is translated into a functionally-equivalent gate-level representation, *netlist*, using a suite of different optimizations and mapping algorithms. Next, in *place-and-route*, the generated netlist is mapped onto the target FPGA's resources (LUTs, FFs, BRAMs, DSP cores, etc). In this step, the routing resources to connect the allocated resources are defined while preserving timing constraints, e.g., operating frequency. The developer can influence the outcome of this step by adding placement constraints as well. For example, the developer can restrict the design to be allocated on a specific region in the FPGA, or can force the routing of connections through specific channels. In *bitstream generation*, the actual binary file that configures the target FPGA is generated. Examples of FPFA toolchains provided by FPGA vendors are Intel Quartus Prime Software Suite and Xilinx Vivado Design Suite.

Partial Reconfiguration enables dynamically reconfiguring a portion of the FPGA, while the rest of the FPGA logic continues to operate seamlessly [37]. An FPGA can be partitioned into a static region and one or more reconfigurable regions (RRs), such that a reconfigurable region can be configured with its own *partial bitstream* without affecting other regions. This has been one of the most significant features of FPGAs in cloud-centric applications, since it allows the cloud service provider to partition the FPGA into one or more RRs executing different functions, thus enabling new features or future updates with increased flexibility and ease. Figure Fig. 1a shows an FPGA configuration with different RRs that can communicate with the CPU or other peripherals via pre-defined interfaces implemented in the static region. As long as the partial bitstream is accommodated by the resources of the allocated RR, the rest of the FPGA does not get affected by the dynamic reconfiguration of that region logic.

3. FPGA Deployment in the Cloud

While early FPGAs were primarily deployed in applications such as telecommunications and ASIC prototyping, more recently, they are used for large-scale datacenters and cloud computing services as *Acceleration-as-a-Service (AaaS)* or *FPGA-as-a-Service (FaaS)*. Both models provide *acceleration* for a wide range of compute-intensive workloads such as machine learning, genomic data processing, and other scientific computations.

In **AaaS**, FPGAs are dedicated to accelerate specific tasks that are pre-defined by the cloud service provider (CSP), e.g., a web search and network encryption. In this model, clients, a.k.a tenants, cannot freely configure the

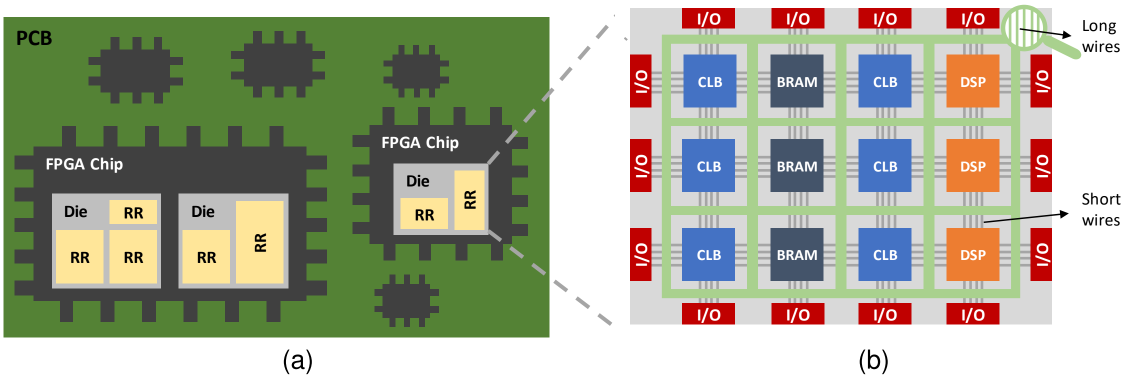


Figure 1. FPGAs on Board: (a) Printed Circuit Board (PCB), Chips & Dies with Reconfigurable Regions (RRs) & (b) Basic FPGA Structure [36].

FPGA fabric as desired, but can only benefit from the accelerated services.

FaaS, on the other hand, enables a more flexible usage where dedicated FPGAs are allocated to tenants and can be freely configured by them as desired. In FaaS, apart from acceleration, FPGAs have also been leveraged in some works [33]–[35] primarily to provide a hardware-based trust anchor for clients in the cloud under minimal trust assumptions. This deployment setting is often referred to as *FPGA-based trusted computing.*

Each of these different deployment settings impose different trust assumptions and requirements. For example, in *AaaS*, since FPGA devices are indiscernible/inaccessible to clients, only the FPGA vendor must be trusted, with no further trust assumptions or guarantees required. In *FaaS*, however, where the clients have more flexible access to the FPGA devices, additional security challenges arise, and thus trust assumptions and requirements differ. Furthermore, effective abstraction, virtualization and resource management techniques (briefly described in § 3.2) are also required.

In this work, we focus on the FaaS deployment model, and investigate the challenges and opportunities in achieving trusted computing in this setting under minimal trust assumptions. We present in the following section § 3.1 more details on the two FPGA-based acceleration deployment models, AaaS and FaaS, adopted by the cloud computing industry. Next, we present an overview on the different virtualization and resource management mechanisms proposed by academia for the FaaS model in § 3.2.

3.1. Commercial Deployment Solutions

FPGA-accelerated computing has been recently deployed by various commercial cloud services providers, e.g., Microsoft, and Amazon AWS. In the following we briefly look into CSPs and their deployment models.

AaaS. Microsoft Azure was among the first datacenters to introduce FPGAs in cloud computing by augmenting CPUs with an interconnected and configurable compute layer of FPGAs to accelerate portions of large-scale software services, e.g., Bing web search [38], or to reinforce the cloud infrastructure by performing network acceleration tasks [39], [40]. Microsoft demonstrated the first proof of concept that deployed FPGAs to accelerate web search ranking on its Bing web search engine [1]. Recently, Microsoft has announced its Project Brainwave [2], a deep learning acceleration platform, aiming to deliver real-time AI. The project aims to support the AI capabilities in Microsoft services like Bing web search and Skype language translations.

FaaS. Several CSPs offer their clients FPGA accelerated computing instances. Prominent examples are Amazon EC2 F1 [3], Huawei FACS FP1 [41], Alibaba F1 & F3 [42], Baidu FPGA Cloud Server [43], and Telekom ECS [44]. The differences among them are mainly in the allocated resources (RAM, vCPUs, network, etc.) and the available FPGA devices a client can rent. Depending on the selected instance capacity, multiple FPGA devices can be configured together to enable larger applications distributed efficiently across the FPGAs. Furthermore, the allocated FPGA devices are dedicated exclusively to the user during the paid period, i.e., spatial multi-tenant FPGA usage is not yet provided, as far as we can infer from the publicly available service description. These instances enable clients to develop FPGA-based services and accelerators by providing a Virtual Machine with pre-installed and licensed FPGA design tools , i.e., Intel Quartus, Xilinx Vivado or SDAccel, depending on the FPGA vendor as well as hardware and software development kits. These kits enable both expert hardware developers and non-specialist developers to generate the desired hardware logic. After the hardware development process is complete, the client uploads the resulting design netlist (the outcome of the synthesis step, cf. § 2) to the cloud. The corresponding FPGA image/bitstream is generated at the CSP side after inspection of the netlist against design rule checks. Then the client proceeds with the software development process, i.e., to develop, debug and run the applications that will benefit from the FPGA-accelerated tasks. Clients can develop their own FPGA-accelerated tasks for personal usage or for commercial purposes, where they can offer their accelerators in the corresponding cloud-based marketplace.

3.2. Academic Deployment Models

In this section we provide an overview of abstraction, virtualization and resource management techniques in the FaaS model as proposed in academic research.

In order to maximize resource utilization and return-on-investment of FPGA resources deployed in the cloud, these

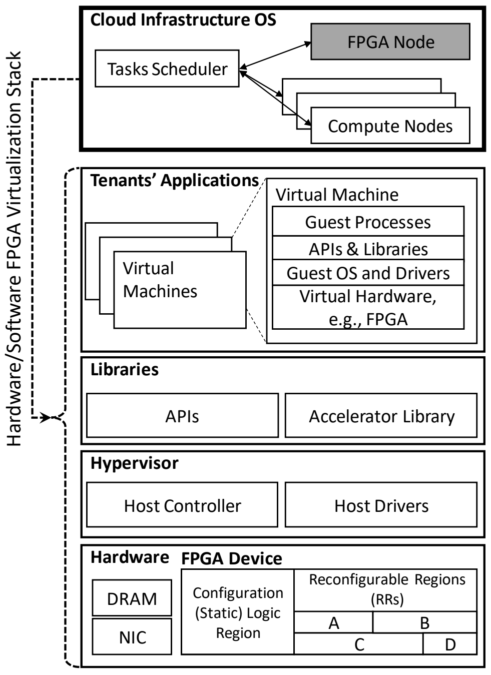


Figure 2. FPGA virtualization in typical cloud computing.

devices can be shared among different tenants, owing to the reconfigurable nature of FPGAs (cf. § 2). Virtualization is used to enable this sharing by abstracting away the complexities and intricacies of the underlying devices and simplifying the interface of interaction with the FPGAs. Earlier FPGA virtualization was primarily concerned with temporal FPGA multiplexing [45] to swap in and out partitions of larger designs when device capacity was limited. Over the years, FPGA virtualization has evolved significantly (in line with evolving FPGA capacities and capabilities), and is currently almost aligned with that of typical CPU and I/O virtualization [46], as shown in Fig. 2.

However, certain FPGA features, e.g., their heterogeneous fabric, execution model and how their logic configuration is vendor-dependent, differentiate them from typical CPU computing. Thus, this imposes specific challenges and requirements on how the virtualization can be handled, as we describe next.

Virtualization. There has been significant progress in FPGA virtualization in recent years [46]. Most of the state-of-the-art work, proposed by academia, focuses on performance boosting for the FaaS cloud-based model [4]–[10]. Nevertheless, both cloud-based models (FaaS & AaaS) require some mechanism of virtualization, similar to that in Fig. 2 to abstract hardware-specific details and represent the FPGA device and its components as a resource pool that can be actively managed, queried, allocated/de-allocated by tenants through the OS deployed in a cloud infrastructure. The cloud OS scheduler handles requests from tenants and creates virtual machines (VMs) for them, and schedules their tasks and resource requirements across a number of available resources and compute nodes, e.g., CPUs, memory, disks, as well as FPGAs. The tenants are thus provided access to their required resources via these VMs.

Granularity of Virtualization. The next question of FPGA virtualization concerns the level of granularity of FPGA primitives that is abstracted into a resource pool available to tenants. Abstracting FPGAs into registers, LUTs, I/O blocks, and memory blocks that can be managed by the cloud OS for tenants to request is, however, not trivial. This is because current FPGA-based development remains largely dependent on the specific hardware fabric of the FPGA and its layout details (cf. § 2). These details must be visible to the configuration toolchain provided in order to generate a compatible FPGA bitstream, where hardware-independent FPGA bitstream configuration remains an open research problem. Moreover, the irregular distribution of these different primitives across the FPGA fabric imposes spatial constraints on how the device gets partitioned into reconfigurable regions (RRs) and how user logic gets mapped to these regions. Recent FPGA devices, however, are attempting to achieve more regular distribution of primitives across the fabric [47].

Virtual FPGAs. Thus, FPGA virtualization schemes usually propose to provide a pre-defined pool of different fixed-size reconfigurable regions (RRs) of the FPGA [4]–[9], e.g. any of regions A - D in Fig. 2. Each reconfigurable region is considered a "virtual FPGA (vFPGA)" where any accelerator can be configured in it, as long as the region provides the required resources, thus providing some degree of independence from the FPGA fabric spatial specifics. Nevertheless, to enable higher flexibility with respect to the sizing of these RRs, recent work [10] has introduced a new layer of abstraction that encapsulates user FPGA logic and enables its mapping to dynamically-sized FPGA physical zones.

FPGA Shell. Besides RRs, some logic on the FPGA must remain statically configured (as shown in Fig. 2), sometimes referred to as the *FPGA shell* or hypervisor logic [6]. FPGA shell provides the physical infrastructure, i.e., clock signals, communication interfaces and reconfiguration management, that actually configures the remaining FPGA fabric (RRs A - D) with the partial bitstreams of the desired accelerator functionalities and controls their connections to FPGA interfaces.

Virtualization Stack. To share the available FPGAs and the RRs therein across multiple tenants as described, a *hypervisor* layer is required to provide host drivers to enable accessibility for the tenants to the underlying FPGAs. A controller module is also required to manage the resources allocation, perform address translation and provide bottom-level software interfaces to the FPGAs, thus providing multiple virtual instances of the FPGA devices [6]. *Libraries* are required to wrap up the available services (different FPGA configurations) into accelerator functions and maintain their bitstreams, while also providing respective APIs for tenants to call and use the accelerators from their software applications.

A tenant can use an API to call an accelerator function either i) from a *pre-defined pool* of different accelerator functions provided by the CSP such that the respective pre-compiled bitstream of the selected accelerator function is used to configure one of the available reconfigurable regions (RRs) [5], [6], or ii) of *own choice* i.e., a tenant may be allowed to provide an FPGA design that gets compiled into a compatible bitstream at the CSP side [5], [6], [9], or even be allowed to directly upload the accelerator's bitstream, given that the tenant is provided with the required information, i.e., vendor-dependent specifics and location on the FPGA to generate the partial bitstream [8].

The FPGA device itself can be either i) installed as a separate acceleration card that is either tied to a host CPU through PCI Express (PCIe) [6], [7], or configured as a stand-alone compute node that is accessed directly and independently over the network only [5], [8], [48] or over both the PCIe and the network [9], or ii) integrated into the same die package as the processor.

Memory accesses by the FPGA to the host's main memory use host physical addresses, whereas VMs on the cloud use guest physical addresses. Hence, an IOMMU is used to translate between the two address spaces to enable accelerators to access VM memory regions to read or write the required data. To further optimize the performance, FPGA memory accesses may be served from the host's last-level-cache (LLC) as well as an FPGA-exclusive local cache, which is kept coherent with the CPU's cache subsystem.[1]

FPGA (Temporal and Spatial) Multi-Tenancy. This virtualized deployment described above enables FPGA multi-tenancy, which traditionally, refers to *temporal* sharing of FPGA devices. Temporal sharing is where the FPGA device itself, or the accelerator functionality configured on it, is used by different users or tenants at different time slots (by some form of time multiplexing), but never simultaneously. This is the more popular FPGA sharing model and the one commonly deployed now in commercial solutions, as described earlier in § 3.1. More recently, however, there has been a growing interest in both academia and industry to enable *spatial* multi-tenancy on FPGA devices as well, where multiple tenants' logic are co-located on the same physical FPGA device simultaneously [49], [50]. In other words, mistrusting tenants would have their configuration bitstreams simultaneously occupying logically isolated RRs, e.g., A and B in Fig. 2. While such a deployment model is principally possible, and would ultimately enhance resource utilization and performance gains, it raises a multitude of challenges that have not been systematically investigated before. One of the key challenges that distinguishes this type of sharing on FPGA from CPU sharing across multiple tenants, is the security issues that arise, both at system- and physical-level, due to the sharing of the physical and raw hardware fabric/compute substrate itself, where the mutually mistrusting tenants have complete privilege and freedom to configure the underlying fabric (within their allocated RRs) as desired. Even if each tenant is restricted to configure the fabric of its allocated RR, the underlying hardware substrate and its physical implementation, e.g. the power supply and distribution are still shared, thus isolation at this level is challenging.

Performance vs. Security Gap. From a performance and efficiency standpoint, there is an abundance of significant works in the literature [4]–[10] that have been concentrated on boosting performance and resource utilization for FPGA-based accelerated computing, i.e., are *performance-centric*, as shown above, and assume a trustworthy CSP. On the other hand, from a security standpoint, the state of FPGA-based trusted computing lags behind. Very little has been invested in investigating and enabling a *security-centric* end-to-end deployment model that leverages cloud FPGAs to perform security-critical tasks on sensitive data, where clients can have their protected IP designs accelerated on cloud FPGAs without having to trust the CSP [33]–[35]. More details on the different adversary models of these different computing settings are presented next in § 4.

1. https://wiki.intel-research.net/FPGA.html

4. System Model & Threat Analysis

In this section we provide an analysis of the stakeholders, threats and resultant security requirements that emerge in FPGA cloud deployment settings.

4.1. Stakeholders

An overview of the of the system model of FPGA cloud computing deployment is shown in Fig. 3. We discuss next the involved stakeholders and their trust assumptions.

FPGA-based cloud services are commonly deployed by a *CSP*, which provides different usage models and services that involve heterogeneous architectures (cf. § 3). This usually involves a standard CPU-based host interfacing with co-processors and accelerator devices, such as GPUs and FPGAs, and offloading particular computation tasks to them as shown in Fig. 3. The CSP, being the platform owner of these facilities, may opt for temporal or even spatial multi-tenancy, i.e., offloading workload from different tenants to an accelerator device simultaneously, to maximize resource utilization and return on their investment, while reaping even higher performance gains.

FPGA vendors provide the FPGA devices and their toolchains to the CSP, which are in turn made available (besides additional development environments also provisioned by the CSP) for the tenants to use over the CSP provided service. FPGA devices and their toolchains are usually assumed trusted by the different stakeholders.

Clients or tenants rent the desired computation capacity and facilities and communicate their workload, which may include security-sensitive data as well as intellectual property (IP) design or code, to the CSP. Therefore, clients sharing resources in the cloud are mutually mistrusting. However, clients may trust the CSP with their sensitive data and only require that the CSP provides isolation among different tenants' workloads. Otherwise, clients may additionally also not trust the CSP, thus requiring it to deploy technologies such as Trusted Execution Environments (TEE) to isolate tenants' workloads from the system software and the CSP itself.

Besides tenants' IPs, the CSP, FPGA vendors and *IP partners* or enterprises, which are developing their own third-party IPs, may release their IPs in an IP marketplace (hosted by the CSP) for tenants to rent and use. To establish secure communication between the clients and the CSP, a *trusted authority (TA)* is required for the management and provisioning of digital certificates, which could either be another dedicated party or the CSP.

Further amplifying the complexity of these relationships, a very recent trend for maximizing datacenter efficiency and flexibility also involves the CSP disaggregating its facilities across multiple different distributed *resource pools* [31], and thus migrating the computation to these resource pools.

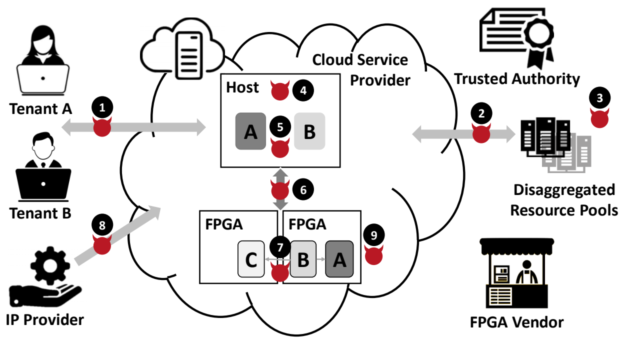


Figure 3. Stakeholders and their security concerns in FPGA-based cloud computing.

Other stakeholders are also involved, which we purposefully keep out of scope to keep the threat modeling more realistic and practical. Examples include foundry facilities that the hardware and FPGA vendors rely on for fabricating their devices, as well as independent entities that issue certificates for CSPs' compliance with security and privacy standards. We assume, however, that trusting these entities, especially fabrication facilities (for the CPUs and FPGA devices acquired by the CSP) is a reasonable and unavoidable assumption. Moreover, such threats and their proposed mitigation mechanisms occur at a very different layer of abstraction, and are an active and orthogonal research focus [51].

4.2. Threat Landscape & Security Requirements

Based on the system model and stakeholders described above, we identify two classes of security threats, demonstrated in Fig. 3. The first includes threats that are also common with traditional CPU-based cloud settings (§ 4.2.1) and do not require the configuration of malicious primitives on cloud FPGAs, while the second class of threats is exclusive to cloud FPGA devices (§ 4.2.2).

4.2.1. Traditional Threats. These are common to cloud-based VM settings and include:

Network Attacks. Tenant IP/data must be protected when transmitted to the cloud, whether a centralized cloud infrastructure ❶ or disaggregated/distributed cloud resources ❷, against an external network adversary.

Physical Attacks. Tenant IP/data must be protected at rest and in processing against an external adversary with physical access. Note that physical attacks are considered a threat only with disaggregation to the edge, where an external adversary may gain physical access to unattended edge devices, as opposed to a single, centralized and physically secure datacenter ❸.

Software Attacks. Tenant IP/data must be protected against potentially compromised hypervisor or system software ❹ and compromised co-tenants' workload on the host ❺. Examples of such attacks are software runtime attacks, microarchitectural and cache side-channel attacks, Rowhammer [52] and software-exploitable physical attacks [53]. We discuss in § 5 how some of these attacks can be also launched from FPGAs.

I/O and System Bus Attacks. Tenant IP/data, which are offloaded to accelerator devices, e.g., GPUs or FPGAs, must be protected when transmitted to/from the accelerator device ❻. This largely depends on, and may also influence, the choice of communication link between the host and the FPGA, e.g., physical link such as PCI Express, Ethernet or a wireless link. Possible attacks include man-in-the-middle attacks, payload misdirection, and device/communication spoofing or sniffing.

Countermeasures. Based on the cloud architecture and the assumed threat model some of these threats may not apply, e.g., physical attacks by the CSP are usually excluded and the CSP is assumed to protect the disaggregated cloud infrastructure against external adversaries with physical access. Other threats may not require explicit catering and assurances, if certain reasonable trust assumptions are instead made. For example, if tenants are willing to trust the CSP with their workloads, security concern ❹ is no longer valid, though standard cloud security measures should still be in place to isolate mistrusting co-tenants' workloads. However, if an untrusted cloud environment is assumed, clients may request to run their workloads in TEEs to isolate and protect co-tenants' workloads from the CSP itself [54]–[56].

4.2.2. FPGA-Specific Threats. This second class of threats stems from the reconfigurable nature of FPGAs and include:

Remote Physical Attacks based on Malicious FPGA Configurations. Tenant IP/data must be protected from untrusted and potentially malicious logic configured on the FPGA device. Damage, which can be caused by malicious logic executing on the FPGA, does not only affect co-tenants on the same physical FPGA but also other resources of the cloud infrastructure in general ❼. We discuss these attacks in detail in § 6 and § 7.

IP Theft or Corruption. Hardware IP providers must be protected from malicious access and counterfeiting, and be able to manage, query and monitor their IP licensing rights and usage ❽. This also includes protecting tenant IP by protecting the confidentiality and the integrity of FPGA bitstreams before and after configuration on the FPGA ❾.

Countermeasures. To tackle FPGA-specific threats, while assuming trusted FPGA devices and toolchains, additional trust requirements should be addressed to achieve secure spatial and temporal multi-tenant FPGA deployment in the cloud. We summarize their potential defenses below:

Defenses Against Remote Physical Attacks. These include i) preventive defenses, e.g., bitstream validation, that mitigate remote physical attacks *before* configuration on cloud FPGAs by detecting logic primitives that cause them (cf. § 8.1), and ii) runtime defenses that address the mitigation gap, which cannot be comprehensively closed by preventive defenses (cf. § 8.2).

IP Protection. Protection of confidentiality and integrity of IP bitstreams as well as secure configuration of bitstreams on cloud FPGAs either via direct or remote interfaces (cf. § 8.3).

In what follows, we describe the aforementioned two classes of attacks in more detail. The first class of attacks common to other cloud computing settings that do not require the configuration of malicious primitives on cloud FPGAs are described in § 5. The second class of attacks, remote physical attacks in multi-tenant FPGAs, are described in § 6 and § 7. The key enabler of the latter attacks is that clients can freely configure their (malicious) hardware circuits on the FPGA. For a better understanding, we provide a brief description of the malicious reconfigurable primitives deployed in these attacks in Appendix § A. These primitives include delay-line sensors, ring-oscillator (RO) sensors and RO power viruses. Most of these attacks assume spatial multi-tenancy (§ 6), however, some of the attacks are also applicable in the temporal multi-tenancy model adopted today in commercial cloud infrastructures (§ 7). All remote physical attacks and defenses are also summarized in Table 1.

5. Traditional Attacks

In this section we briefly outline attacks that are not FPGA-specific, i.e., they do not require the configuration of malicious primitives on FPGA logic in the cloud. These attacks are, thus, applicable in both FPGA deployment models (temporal or spatial) and have their equivalents in CPU- and GPU-based computing. They usually exploit non-configurable hardware and microarchitectural features that already exist in both computing systems, e.g., shared caches and DRAM-based main memories. Therefore, we also argue that the established defenses to mitigate these attacks in CPU-based computing can also be applied for FPGA devices, and we mention them briefly.

Rowhammer Attacks. Weissman et al. [74] demonstrated a software-based fault injection attack that leverages the Rowhammer effect in DRAM memories. By excessively accessing specific rows in the DRAM memory bit flips in neighboring DRAM cells can be induced [52]. The attack is demonstrated using two different setups[2], Intel Arria 10 FPGA integrated into the CPU package (A prototype E5-2600v4) and Intel Arria 10 acceleration card. Compared to a Rowhammer attack conducted by the CPU, the FPGA can achieve faster memory accesses and thus more bit flips due to the fact that, unlike memory accesses from the FPGA, memory accesses from the CPU must be followed by flushing the cached data. To conduct the Rowhammer attack, the FPGA is configured to perform memory accesses for specific amount of time, the parameters (target memory addresses and number of times to be accessed) are provided by the CPU.
Mitigation. As a countermeasure against these Rowhammer attacks, in line with these proposed for CPU-based attacks, the authors propose to increase DRAM row refresh rate. Another solution would be monitoring and detecting excessive DMA commands by the FPGA shell [6].

Cache Attacks. The feasibility of cache attacks on both the integrated Arria 10 FPGA and the Arria 10 acceleration card have also been shown by Weissman et al. [74] since the CPU and FPGA share memory and the last-level-cache (LLC). More specifically, the Arria 10 acceleration card has its own local DRAM besides access to that of the CPU. When the card reads from the CPU's memory, the request is served from the CPU's LLC if possible, otherwise from the main memory. Thus, the card is able to influence the state of the LLC and place cache lines into it, but only by writing (not reading) to the corresponding memory addresses. The integrated Arria 10 FPGA has access to the CPU host memory, and has its own dedicated directly-mapped local cache that is kept coherent with the CPU's cache. Depending on the type of memory access, it is either served directly from the CPU's LLC and main memory or it is first checked in the FPGA local cache. Then on a miss, the request gets forwarded to the CPU's LLC and main memory. Owing to both shared cache and memory, different cache-based attacks can be mounted by either the FPGA or the CPU to target the CPU's LLC or the FPGA local cache respectively. While eviction-based attacks can be mounted from the FPGA to influence and prime the CPU's LLC successfully, flush-based attacks (that require the `clflush` instruction) are not possible since the FPGA cannot run this instruction. On the other hand, eviction-based attacks that require to prime the FPGA local cache from the CPU are not possible, though flush-based attacks that leverage the `clflush` instruction are possible since the FPGA local cache is kept coherent with the CPU's cache. Furthermore, a covert channel can be constructed from the FPGA logic to the CPU by manipulation and probing of the LLC state.

Similarly, cache attacks in spatial multi-tenant FPGAs, where both the victim and attacker are co-located on the same FPGA device, would also be easily feasible. These attacks would exploit the FPGA local cache, so long as it is unconditionally shared among the different tenants, similar to how CPU cache is shared among different processes or cores. If the FPGA local cache is a directly-mapped cache, e.g. as in the Arria 10 FPGA, eviction set construction becomes trivial, especially since FPGA logic is usually allocated memory pages such that the requested memory buffer is within contiguous physical memory.
Mitigation. Cache partitioning mechanisms proposed to protect CPUs against these attacks, either software-based [75]–[77] or hardware-based [78]–[82], would, in principle, also mitigate cache attacks on FPGAs. However, to mitigate cache attacks in spatial multi-tenant FPGAs, where the CPU host OS is not involved, hardware-based defenses are the only feasible approach. Cache partitions would be utilized exclusively by the co-tenants, thus enforcing cache isolation among them and blocking side channels. Alternatively, specific cache lines that are security-critical can be tagged and locked in cache [78], [80] by hardware-based mechanisms, and thus cannot be evicted by other co-tenants. Hardware-based randomization defenses [83]–[85] proposed for cache attacks in CPU computing would also mitigate them with similar guarantees on FPGA devices.

6. Spatial Multi-Tenancy: Remote Physical Attacks

In spatial multi-tenant deployment model, a tenant that has the freedom to configure malicious primitives (cf. § A) on one of the virtual FPGAs, i.e., reconfigurable regions, can launch a physical attack on a victim logic occupying a virtual FPGA on the same physical FPGA as the attacker. For a better understanding of the attacks, we provide a brief description of the power supply system of an FPGA device next and classify the physical attacks based on the exploited physical phenomena into power leakage attacks § 6.1 and crosstalk effect attacks § 6.2.

Power Distribution Network (PDN) for an FPGA consists of a voltage regulator, decoupling capacitors and

2. https://wiki.intel-research.net/FPGA.html

TABLE 1. Remote Attacks and Defenses at a glance.

	Leakage Source			
Remote Physical Attacks	Power Leakage	Power Drop	Crosstalk	Thermal Leakage
Spatial Multi-Tenancy	[15]–[19], [21], [22]	[13], [14]	[23]–[27]	-
Temporal Multi-Tenancy	[28], [29]	[11], [12]	-	[30]
Defenses				
Bitstream Validation	[57], [58]			
Other Preventive Solutions			[25], [26], [59]–[61]	
Runtime Solutions	[20], [62]–[70]	[71], [72]		[30], [72], [73]

a grid of interconnects, modeled as an RLC-network, to deliver the power and ground voltages to FPGA components [86], [87]. The voltage drop V_{drop} is modeled as the sum of two components a *steady-state* voltage drop IR, caused by the interconnects' resistance of the RLC-network, and a *transient* voltage drop Ldi/dt, caused by the interconnects' inductance of the RLC-network. The power consumption is a product of voltage drop and drawn current. Since the voltage supplied to the FPGA device must remain fixed at a pre-defined level, changes in power demands, which vary based on the running functionality, are manifested as changes in current demands. The voltage regulator and the decoupling capacitors operate in parallel to accommodate the changes in current demand and to compensate steady-state and transient voltage drops, respectively, to try and maintain a fixed voltage level. However, in case of sharp transient changes in current demand of the FPGA, neither the voltage regulator nor the decoupling capacitors can respond fast enough. This leads to spatial and temporal fluctuations of the voltage level in the PDN, which may result in functional failures.

6.1. Power Leakage & Power Drop Attacks

Power consumption of victim cryptographic cores, configurable or running as a software (SW) process on an FPGA-based SoC, can be measured using either one of the configurable voltage sensors to retrieve their secret keys. Further, power leakage can be leveraged to construct a covert channel between malicious co-tenants. In the following we categorize state-of-the art attacks into side and covert channels.

Side-Channel Attacks. Schellenberg et al. [15] deployed a delay-line sensor to detect voltage fluctuations in the PDN caused by the activity of a soft AES-128 core, both located on the same FPGA fabric and logically isolated by passive fence of unused logic [88]. Using correlation power analysis (CPA), encrypted messages and the measured sensor values, i.e., power traces, the secret key can be recovered. The attack, which is demonstrated locally (in controlled Lab settings) on a SAKURA-G board with Xilinx Spartan 6 FPGAs, is shown to be successful for different locations of the sensor with reference to that of the AES core, but fewer traces are required when the sensor is closer to the AES core. The same attack was conducted on AES using several RO-based sensors in [16].Further, the attack is deemed feasible on Xilinx UltraScale+ FPGAs deployed in Amazon F1 instances. In [17] the AES key was fully recovered in 42% of 30 attempts made on different Amazon F1 instances.

In [18], RO-based voltage sensors are leveraged to monitor the activity of a soft RSA-1024 core, based on square-and-multiply, co-located on the same FPGA fabric. Using simple power analysis (SPA) and the sum of power traces from 20 RO-based sensors, RSA private key can be fully recovered. We summarize the details in Table 2. Further, the authors demonstrated that the start of RSA operations cannot be identified from the power traces when a noise generator, whose area overhead and power consumption are comparable to that of the RSA core, is activated. The attack also works when the RSA encryption runs as a process on a CPU that shares the same PDN with the FPGA, i.e., in an FPGA-based SoC. The attacks are demonstrated locally on a Zedboard deploying Zynq-7020 SoC of ARM Cortex-A9 and a Xilinx Artix 7 FPGA.

Similarly, Gravellier et al. [19] demonstrated a CPA attack on AES-128, running as a bare metal program on the CPU of a Zynq-7000 SoC, using delay-line sensors, implemented on the Artix 7 FPGA. In Table 2 we summarize the details of the aforementioned attacks.

Krautter et al. [20] investigated the effect of different placement strategies (default, area-, performance- or power-optimized) as well as the effect of process variation (PV), within a single FPGA instance (intra-PV) and among different FPGA instances of the same family (inter-PV), on the number of required traces by a CPA attack. The authors further showed a difference in the minimum amount of traces required for a successful attack of more than 100x for different locations of the AES core and the sensor on a Virtex 7 ADM-PCIE-7V3 accelerator card, which stresses the great impact of process variation on this attack.

Covert-Channel Attacks. By leveraging groups of RO-based power viruses as transmitters and RO-based voltage sensors as receivers, Giechaskiel et al. [21] demonstrated a one-direction covert channel between two users spatially sharing a cloud FPGA. Although spatial multi-tenant FPGAs are not yet adopted in real-world cloud solutions, such covert channels are shown feasible on Amazon F1 and Huawei FP1 instances, which deploy Xilinx *3D* Viretx UltraScale+ FPGAs. A Virtex UltraScale+ FPGA can accommodate up to four dies called super logic regions (SLRs) in a single chip to deliver high logic density [89], where the SLR dies are aligned in one dimension and share power and clock delivery systems. The authors envisioned, in a spatial multi-tenant scenario, that each SLR would be dedicated to a different tenant and demonstrated the construction of a covert channel between two SLRs based on power leakage, which is confirmed to be stronger when the receiver and the transmitter occupy neighboring SLRs. The attack is demonstrated on local and cloud Virtex UltraScale+ FPGAs (Huawei FP1 and Amazon F1 instances). In Table 3 we summarize the required resources and the performance (accuracy and bandwidth) of the attack. Different bandwidth numbers (bits per second)

TABLE 2. SPATIAL MULTI-TENANCY: REMOTE SIDE-CHANNEL ATTACKS LEVERAGING POWER LEAKAGE.

Ref.	Crypto Core			Sensor Properties		Required	local	Analysis
	HW/SW	Data Path	Freq.	Type	Sampling Freq.	Encryptions	/cloud	
[19]	bare metal SW process	AES 32-bit	667 MHz	8 delay-lines (5-bit TDC)	200 MHz	87K	local boards	CPA
		AES 8-bit				111K		
	HW	AES 128-bit	10 MHz			4.5*K*		
[15]		AES 32-bit	24 MHz	1 delay-line (6-bit TDC)	24 MHz [a]	5K [b]		
[17]		AES 128-bit	8 MHz	1 delay-line (8-bit TDC)	96 MHz	500K	Amazon F1	
[16]		AES 128-bit	50 MHz	64 ROs & 8-bit counters	250 MHz	8K	local boards	
[18]		RSA 1024-bit	20 MHz	20 1-stage ROs & 14-bit counters	0.1 MHz	9		SPA

[a] Results apply also to other sampling frequencies: 48, 72 & 96 MHz.
[b] The results correspond to the key bit showing the highest correlation.

are due to the different operating frequencies: 300 MHz, 200 MHz and 125 MHz on local VCU118, Huawei FP1 and Amazon F1. Transmitter ROs, which are grouped into subsets and placed in different clock regions, are enabled and disabled simultaneously. The transmission between the transmitter and the receiver is based on the Manchester encoding scheme, in which a single bit of information is represented as a tuple: '1' as (1,0) and '0' as (0,1).

Similarly, Gnad et al. [22] demonstrated a power covert channel, however, using a different encoding scheme. To send a '1' the transmitter generates positive voltage spike in the PDN by gradually activating groups of the RO-based power viruses and disabling all of them at once. Whereas to transmit a '0' a negative spike in the power trace should be generated by activating all power viruses at once and gradually disabling them. The receiver is a delay-line sensor that can measure the emerging voltage spikes in the PDN. 25 clock cycles separate between the transmission of two bits to allow the PDN to recover. The covert channel is evaluated on different boards: Pynq-Z1 with Zynq-7020 SoC (Xilinx Artix 7 FPGA) and KC705 with Kintex 7 FPGA. We present in Table 3 the best performance results when the transmitter and the receiver operates at 200 MHz and no additional noise source is running on the Kintex 7 FPGA.

Fault Injection Attacks. Krautter et al. [13] showed that by precisely controlling a power-hungry circuit, which comprises thousands of RO-based power viruses and occupys 30-50% of FPGA LUTs of a Terasic DE1-SoC board deploying Intel Cyclone V, it is feasible to induce timing failures in the 9^{th} round of a soft AES-128 core, i.e., implemented in the FPGA fabric. The induced timing faults are used to recover the secret key by means of a differential fault analysis (DFA) [90], where the victim AES core operating at 111 MHz and spatially shared the FPGA with an array of RO-based power viruses. The achieved key recovery rate is of 93% on average using 18K encrypted messages. Mahmoud et al. [14] also leveraged a power-hungry circuit, occupying 25% of FPGA LUTs of a VC707 board with Xilinx Virtex 7, to induce timing failures during the operation of a true random number generator that are used for the generation of secret keys. The faulty output was shown to be biased and failed randomness tests.

6.2. Crosstalk Attacks

The crosstalk effect is caused by the electromagnetic coupling that occurs between unshielded wires that are laid in parallel. Thus, a signal transmitted through one wire influences the signal propagation delay in neighboring wires. The crosstalk effect is leveraged to launch remote side- and covert-channel attacks.

Side-Channel Attacks. Ramesh et al. [23] exploited the crosstalk effect within long wires in FPGA (cf. Fig. 1b) to recover the secret key of a soft AES-128 core. The transmitter, routed through long wires, is an input to an AES S-box. The snooping circuit comprises a RO-based voltage sensor. The receiver, which is one of the RO's wires, is routed through long wires and is located next to the transmitter wire. Using differential power analysis (DPA) [91], the counter values obtained from the snooping circuit (power traces) and the encrypted messages, the key of the final round of the AES can be recovered byte by byte. Then the encryption key can be obtained by inverting the key schedule. The crosstalk effect increases for longer transmission pair and lower operating frequencies of the victim logic, which directly influences the number of encryption operations required to recover the key. The attack is evaluated locally on several boards of Intel FPGA devices: DE2-115 board, Cyclone IV and Stratix V development kits [23] and DE5a-Net Arria 10 GX boards [24].

The same effect is also demonstrated in [25], where the authors leverage a sliding-window sampling scheme to sample the RO-based sensor at overlapping time periods in order to extract the value of each transmitted bit of a secret key. The authors show that the probability of correctly recovering a key is higher for larger keys (asymmetric vs. symmetric keys) and smaller window sizes. In Table 4 we summarize the performance of each of the aforementioned DPA and sliding window approaches. Long wire leakage is characterized for different generations and families of Xilinx FPGAs, including Virtex 5, Virtex 6, Artix 7, Spartan 7 and Virtex UltraScale+, on local boards [26] as well as on cloud FPGA instances [27]. More technical details on the respective attacks can be found in Appendix § B.

Covert-Channel Attacks. Besides side-channel attacks, the crosstalk effect can be exploited to construct a covert channel between two circuits sharing the FPGA fabric at the same time [25], [26]. Crosstalk-based covert channels have been shown locally on several boards (of Xilinx FPGA devices) in [26]: ML509 (Virtex 5), ML605 (Virtex 6), ArtyS7 (Spartan 7), Nexys 4 DDR and Basys 3 (Artix 7). In Table 3 we summarize the performance of a crosstalk covert channel conducted with transmission pair of 2-segment long wire and Manchester encoding

TABLE 3. SPATIAL MULTI-TENANCY: REMOTE COVERT-CHANNEL ATTACKS.

Ref.	Deployed Effect	Bandwidth	Error Rate	Platform	Transmitter	Receiver
[21]	Power Leakage	1.17 Mbps	0.1%	local VCU118	4K 2-stage ROs (Fig. 4c)	25 7-bit counters & 2-stage ROs (Fig. 4c)
[21]	Power Leakage	781 Kbps	0.1%	Huawei FP1	6K 2-stage ROs (Fig. 4c)	
[21]	Power Leakage	488 Kbps	0.1%	Amazon F1		
[22]	Power Leakage	8 Mbps	0	local KC705	5K 1-stage ROs (Fig. 4c)	6-bit delay-line
[26]	Crosstalk	6.1 Kbps	0.1 - 1%	local boards	one signal buffer	single 21-bit counter & 3-stage RO (Fig. 4b)

TABLE 4. SPATIAL MULTI-TENANCY: REMOTE SIDE-CHANNEL ATTACKS LEVERAGING CROSSTALK EFFECT.

	Victim Circuit			Snooping Circuit	Required	
Ref.	Key Size	Operating Freq.	Transmitter Length	Type / Sampling Freq.	Encryptions	Analysis
[23]	AES (128-bit)	1 MHz	1-segment long wire	RO / 1 MHz	233K [a]	DPA
[26]	32-bit	0.78 MHz	$\geq$ 1-segment long wire	RO / 195 KHz	20K [b]	Sliding Window ($w = 4$)

[a] Reported results are for automatically placed and routed victim AES.
[b] For all four counters, i.e., 5K per counter, to recover only 98.4% of the 32-bit key.

scheme without error-correcting codes. Compared to a covert channel based on power leakage, crosstalk effect has notably lower performance (accuracy and bandwidth), however, it also requires less FPGA resources.

To summarize, exploiting the crosstalk effect requires more measurements or traces than a power side-channel attack, as seen in Table 2 and Table 4. Further, exploiting crosstalk effect requires close proximity between the victim and snooping circuit with a maximum distance of one long wire separating the transmitter and the receiver, which is hard to achieve in a real-world deployment model, given that a malicious tenant has no influence on the process of place-and-route of the victim netlist on the FPGA. Moreover, It is demonstrated that the crosstalk effect diminishes for victim circuits operating at higher frequencies [23]. In general these attacks, whether based on power leakage or crosstalk effect, are demonstrated on cryptographic cores operating at low frequencies, e.g., 24 MHz for the AES core, as opposed to the purpose of leveraging FPGAs to achieve the high performance these platforms can offer.[3] Running the cryptographic cores at higher frequencies of few hundreds MHz would require higher sampling rates that are harder to achieve with reconfigurable sensors.

7. Temporal Multi-Tenancy: Remote Physical Attacks

In this section, we present remote physical attacks that assume the conventional temporal multi-tenant FPGA deployment model, which is adopted in real-world cloud infrastructures. That is the whole physical FPGA is allocated to one client at a time. Nevertheless, different tenants can use different FPGA instances (and other resources) at the same time. We classify these attacks into power leakage attacks § 7.1 and thermal leakage attacks § 7.2.

7.1. Power Leakage & Power Drop Attacks

Side-Channel Attacks. Schellenberg et al. [28] deployed a delay-line sensor in one FPGA to measure the power consumption of a soft cryptographic core (AES-128 or RSA-224) implemented on another FPGA that is integrated on the same board and sharing the power supply system. To enable the comparison of results with spatial power side-channel attacks (cf. § 6.1), the authors replicated their settings from [15] (same board and same AES core). They showed that to recover AES key, 2.5 million encryptions are required when the AES core and sensor are on two different FPGAs, compared to 5K encryptions when they are co-located on the same physical FPGA. The attacks are demonstrated on a SAKURA-G board integrating two Xilinx Spartan 6 FPGAs.

Covert-Channel Attacks. Power leakage is further exploited on three realistic setups of FPGA-to-FPGA, CPU-to-FPGA and GPU-to-FPGA covert channels, where transmitters and receivers are separated on two distinct boards sharing only the external power supply unit in [29]. Nevertheless, to capture the power leakage across boards, additional set of RO-based power viruses, the stressor, is required. The stressor power viruses and the receiver sensors, which are on the same FPGA, are grouped into subsets and distributed across different clock regions in the corners and the center of the receiver FPGA, while the transmitter is different based on the source type, i.e., FPGA or CPU/GPU. In case of FPGA-to-FPGA the transmitter is a set of RO-based power viruses, whereas in case of CPU-to-FPGA and GPU-to-FPGA the transmitter is a stress program that runs simultaneously on available threads or cores. During the transmission of one bit, the stressors are alternately enabled and disabled in equal intervals and the counter values of each receiver RO for the enabled and disabled periods are averaged and compared to extract the transmitted bit. Then the final transmitted bit is computed based on the majority of all receiver ROs. The covert channel is demonstrated locally on different FPGA boards KC705 (Xilinx Kintex 7) and AC701 (Xilinx Artix 7), two Xeon processors with 4 threads and 24 threads, and two GeForce GPUs with 96 and 640 CUDA cores. In Table 5 we summarize the best performance results (bandwidth and accuracy) and the required resources to achieve them.

By comparing the results in Table 3 and Table 5 in terms of resources and performance, it is obvious that power covert channels in spatial settings are stronger and require less resources.

3. For example, AES can run at 300MHz on a Virtex 6 FPGA https://opencores.org/projects/tiny_aes.

Denial of Service (DoS) Attacks. Gnad et al. [11] demonstrated that excessive switching activities of a power-hungry circuit, occupying 12-13% of the FPGA LUTs on Xilinx ML605 (Virtex 6), KC705 (Kintex 7) and Zedboard (Artix 7), in the picosecond regime create voltage emergencies in the PDN. Such sharp and transient voltage drops cannot be compensated by the decoupling capacitors or the voltage regulator and cause the FPGA to crash in less than 1 ms. The DoS attack is demonstrated on local boards and is deemed feasible on Xilinx UltraScale+ FPGAs similar to those deployed in Amazon EC2 F1 instances [12]. Note that the size of the power-hungry circuit and the required resources on the FPGA to induce the required effect differ based on the targeted FPGA.

7.2. Thermal Leakage Attacks

Tian et al. [30] demonstrated how to exploit thermal leakage of cloud FPGAs to establish a covert channel using single FPGA. This requires a pre-agreement between the clients on the identity of target FPGA instance, i.e. the communication medium, which implies disclosing information about the cloud infrastructure, i.e., identifying FPGA instances offered by the CSP. One approach to acquire such information is by fingerprinting cloud FPGAs using physically unclonable functions to measure unique characteristics of the FPGAs. Tian et al. [92] conducted the experiments on Amazon AWS cloud and demonstrated the feasibility of renting the same FPGA by two successive clients for the different available instances.

By leveraging a grid of ROs as a transmitter (heater) and RO-based thermal sensors as a receiver, Tian et al. [30] demonstrated a covert channel between two clients temporally sharing the same cloud FPGA. A transmitter circuit is configured on the FPGA and enabled to achieve a steady-state temperature corresponding to logic '1'. Then the transmitter circuit is disabled and the FPGA is left idle before configuring the receiver's bitstream on the same FPGA. The receiver circuit then measures the ROs frequencies and compares them to nominal ROs frequencies which correspond to logical '0' to acquire the transmitted value. The covert-channel bandwidth is defined by the heating period, the maximum time gap between heating the FPGA and sensing its temperature, and the transmitter's size (number of ROs). The longer the idle gap/period between heating and sensing, the weaker the covert channel is. The evaluation is performed on cloud FPGA instances, where for an idle period between two users of a cloud FPGA of 120 seconds, 6.64 minutes are required to transmit one bit. Unlike covert channels based on crosstalk or power leakage, thermal leakage impose no constraints on the placements of transmitter and receiver circuits since the generated heat can be sensed from anywhere on the FPGA. The covert channel is demonstrated on Stratix V FPGAs deployed in Texas Advanced Computing Center. Table 5 shows the performance of the thermal covert channel and a comparison to power covert channels.

8. Defenses

We classify state-of-the-art defenses against remote physical attacks (cf. § 6 and § 7) into preventive solutions that mitigate the threats before configuration on the FPGA (§ 8.1) and runtime solutions that aim to address the mitigation gap that static approaches cannot comprehensively close (§ 8.2). In § 8.3 we examine IP/data protection and secure configuration solutions.

8.1. Preventive Solutions

8.1.1. Bitstream Validation. This attempts to verify that a user design is harmless, i.e., contains no malicious primitives that can be deployed for remote physical attacks. One way to do this is by leveraging design rule checks (DRCs), which are provided in FPGA vendor toolchains, e.g., Xilinx Vivado DRCs, to verify that a design does not violate a set of pre-defined rules. Amazon AWS enforces DRCs on clients' netlists to prevent combinatorial loops, e.g. ROs, before generating the final bitstream. [4] However, alternative sequential RO designs (cf. § A.1) can still bypass the check [27], [93], [94]. In particular, latch-based and flip-flop based RO sensors are used to characterize long wire leakage on Amazon F1 and Huawei FP1 instances [27].

Another approach for bitstream validation is by using stand-alone virus scanner tools [57], [58] that search for uncommon structural and behavioral properties of malicious logic in the netlist. For example, FPGADefender [58] is designed to detect combinatorial and sequential self-oscillating circuits, whereas the virus scanner in [57] is designed to detect combinatorial loops as well as delay-line sensors. Such tools are envisioned to be deployed by the CSP to scan the uploaded FPGA circuits prior to configuration on the FPGA. In case the CSP allows clients to directly upload their bitstreams, reverse-engineering of the bitstream is required to recover the netlist. Moreover, the virus scanner tool must be updated regularly with formulation of properties, i.e., virus signatures, to account for new malicious logic designs.

Both DRCs and virus scanners require access to plaintext client IP designs, which introduces a conflict with IP confidentiality (discussed further in § 9.1).

8.1.2. Prevention of Crosstalk Attacks. Giechaskiel et al. [25], [26] proposed to block four neighboring long wires, two long wires from each side, of the victim wires. The blocked wires can be left unoccupied or further driven by random values. This can be performed either by manual inspection of the placed and routed design to identify the source of leakage and add the guarding wires or by the support of FPGA toolchains. This requires including directives in the source code of the design by the developer to annotate sensitive signals and further modifying routing algorithms to automate adding the guarding wires.

A compromise solution is proposed by Luo and Xu [60]. They implemented a framework in the command line interface of FPGA toolchains to prioritize the placement of sensitive logic based on their security-levels in a spiral manner such that the most sensitive logic is in the center of the allocated blocks. For example, the S-Boxes of the AES core are placed in the center of the allocated logic and are routed away from long wires. External interfaces to the sensitive logic, e.g., UART or PCIe, routed through long wires are guarded by driving the neighboring long

4. https://github.com/aws/aws-fpga/blob/master/ERRATA.md

TABLE 5. Temporal multi-tenancy: Remote covert-channel attacks.
All attacks use 3-stage RO designs (RO Fig. 4a in [30] and RO Fig. 4b in [29])

Ref.	Deployed Effect	Bandwidth	Error Rate	Platforms	Transmitter	Receiver	Stressor
[29]	Power	6.1 bps	$\leq 5\%$	KC705-to-AC701	14K ROs	20 ROs & 15-bit Counters	500 ROs
[29]	Power	6.1 bps	3%	CPU-to-AC701	14 threads		
[29]	Power	2 bps	3%	GPU-to-AC701	640 CUDA core		
[30]	Thermal	1 bit per 6.64 minute	0%	Altera Stratix V	41K ROs *	RO sensors *	N.A.

* Exact numbers are not given.

wires with '0', '1' or random values. The framework operates at the netlist level of the design, therefore, it is unlikely that additional resources (LUTs or memory blocks) will be incurred. Nevertheless, constrained place-and-route would affect the maximum operating frequency of the circuit, besides the two additional long wires, surrounding each sensitive long wire, that are reserved and driven by obfuscation logic.

Seifoori et al. [61] demonstrated that modifying the PathFinder FPGA routing algorithm eliminates the crosstalk effect in the context of integrating untrusted third-party IP cores and security-critical IP cores in a single design. The authors proposed four routing strategies that either i) entirely block the nearest two or four neighbors of a sensitive wire, ii) block only untrusted signals (of a different core) from occupying the two neighbors of a sensitive wire or iii) allow only trusted signals (of the same core) to occupy the two neighbors of a sensitive wire. The performance loss is evaluated for each strategy in terms of routing channel width and critical path delay (maximum operating frequency).

Xilinx Vivado toolchain includes an Isolation Design Flow, which enforces physical separation between different cores/designs on the FPGA, such that the entire design (functions & signals) is placed-and-routed inside the defined region [59]. Then, Vivado Isolation Verifier can be used to ensure that isolation rules are met. This mitigates the crosstalk leakage, however, it may introduce performance issues due to routing constraints.

In summary, mitigating crosstalk should be adopted by the CSP to secure both clients logic and their interfaces to the FPGA shell in case clients are only allowed to upload their designs/netlists. Otherwise, clients should make sure that their circuits are not routed through long wires, whereas the interfaces to the FPGA shell should be still secured by the CSP.

8.2. Runtime Defenses

We refer here to runtime solutions that aim to i) detect 'malicious' activities by detecting changes in operating conditions or ii) prevent or harden the exploitation of the leakage during the execution on the FPGA.

8.2.1. Detection of Power-Drop Attacks. Provelengios et al. [71] proposed to deploy a network of RO-based sensors in the FPGA shell to periodically detect and identify malicious voltage drops and to locate the center of a power-hungry circuit. This information is used by CSP to revoke the malicious logic from the FPGA. The accuracy of this approach depends highly on the number of sensors and also the intensity of the attack (number of power viruses).

The more sensors and the intenser the attack, the more accurate the results will be. In their settings, the authors used 46 RO sensors (each comprises a 19-stage RO & a 20-bit counter) resulting is 5% resource utilization on a Intel Cyclone V FPGA of DE1-SoC board. Although the authors do not elaborate on the time required to locate the center of the malicious logic, it is very likely that until the malicious logic is located and revoked, timing errors in neighboring logic would still occur or the FPGA would be shut down. Alternatively, when voltage drops are detected, the clocks of virtual FPGAs can be paused until the CSP locates the source of the malicious activities. Note that the clients need to trust the CSP, otherwise, a malicious CSP can deploy a sensor network to spy on the clients.

Xilinx provides a security monitor IP core that can be instantiated in some of their FPGAs [72]. This IP core monitors supply voltage and temperature in addition to other functionalities and allows the user to decide among different configurable penalties. Such IP core can be instantiated by the CSP to detect voltage drop in the FPGA and zeroize (regions of) the configuration memory for example. However, it is not clear whether it can provide information about the source of the voltage drop.

8.2.2. Mitigation of Power Leakage Attacks. Krautter et al. [62] proposed a hiding countermeasure, through which side-channel leakage is reduced by implanting a fence of ROs that works as a noise generator circuit around the logic to be protected, thus reducing signal-to-noise-ratio (SNR). The fence has a size of roughly the size of of the design to be protected and its power consumption is slightly more than half of the AES power consumption. Krautter et al. [62] further proved that a fence activation strategy based on internal power measurements, using voltage sensors, is more effective than a random activation pattern. Their approach increased the number of required traces to recover the key from 1.8K, when no defense applied, to 300K for sensor-based activated fence. Nevertheless, physical locations of the fence and AES core highly affect the size of the fence and the required traces for a CPA attack. Krautter et al. [20] showed that by carefully choosing the AES core location on the targeted FPGA, the same protection level of the active fence approach can be achieved with zero overhead. However, this requires exhaustive per-device analysis to find the least vulnerable location per FPGA.

Note that implementing an active fence by a client would not be allowed if a bitstream validation process is applied by the CSP. On the other hand, if such a defense is to be applied by the CSP, then the CSP needs to orchestrate active fences for the virtual FPGAs, resulting in additional and non-trivial overhead in terms of power and FPGA resources. Other hiding techniques based on dual-rail precharge logic or duplication schemes [63], [64] can be implemented directly by the clients in their allocated virtual FPGAs. However, they still suffer information leakage due to imbalanced routing on FPGAs and result in duplicated area and power overhead.

Other countermeasures that can be implemented by the client against power leakage on FPGAs are randomization and masking. Randomization approaches work by randomly changing operating frequency (if allowed by the CSP) [66], [67], [70], execution of random dummy operations [66] or adding random delays to input signals [68]. Masking techniques for reconfigurable platforms are either inefficient with respect to area and performance [69], or still generate exploitable leakage [65]. Moreover, masking techniques should be designed for every cryptographic function individually and the desired security guarantees.

8.2.3. Mitigation of Thermal Leakage Attacks. Intel integrates diode temperature sensors in the configurable fabric of their FPGAs [73]. To mitigate thermal leakage, these sensors can be exclusively leveraged by the CSP to monitor the temperature across the device at runtime. Once the temperature reaches a pre-defined level, the CSP can apply a suitable cooling countermeasure.

In [30] the authors proposed enforcing a minimum idle period, in the order of minutes, for the FPGA between two successive allocations. This allows the FPGA to recover to a nominal temperature. Alternatively, they proposed to heat or cool the FPGA to a pre-defined temperature before it is reallocated. This mitigation needs to be considered for practical feasibility and against the CSP's drive for operational efficiency where idle cycles are lost income.

8.3. FPGA IP & Data Protection

We present in the following an overview of academic solutions that are focused on IP protection in an untrusted cloud environment for FPGA-based *trusted computing*. Then we refer to other schemes that try to address data isolation and access control for cloud FPGAs.

8.3.1. IP Protection. State-of-the-art solutions rely on an initial bitstream that configures a trust anchor on a cloud FPGA. The main goal of this trust anchor is to run a key exchange protocol with clients and decrypt client' IP bitstream prior to configuration on the FPGA. The trust anchor contains a cryptographic core for decryption of IP bitstreams using secret session keys, which are obtained through the key exchange protocol that deploys public key cryptography, e.g., RSA or elliptic core cryptography (ECC) [33], [35], [95]. In [33], the authors proposed to embed the private key directly with the RSA core in the initial bitstream, thus, the confidentiality of the initial bitstream must be protected and this is achieved, e.g., by leveraging the built-in AES engine in Xilinx FPGAs. For that, the authors propose a trusted party, e.g., the FPGA vendor, to program the AES secret key before deploying the FPGA in the cloud. In [95], the authors do not protect the initial bitstream but rather obfuscate the public key into a finite state machine (FSM) [96]. Such that the public key cannot be inferred by static analysis of the initial bitstream, i.e., the initial bitstream must be configured on the FPGA in order for the trust anchor to generate the public key. The security of the scheme relies on strong assumptions of time-bounded protocol responses to detect network attacks, e.g., man-in-the-middle. Moreover, the unprotected initial bitstream is vulnerable to bitstream manipulation attacks [97]. In [35], the authors embed a physically unclonable function (PUF) in the trust anchor to generate device-specific session keys. This implies protecting only the integrity of the initial bitstream as PUF outputs are generated on-the-fly. However, the authors instead assume that the FPGA vendor configures the trust anchor on the FPGA prior to deployment in the cloud and that the FPGA is constantly powered, even during shipping to the CSP, to maintain its configuration. Alternatively, the initial bitstream of the trust anchor can be protected and stored in off-chip memory as in [33]. Note that the trust anchor, which is designed and implemented by the trusted party, may also include the rest of the FPGA shell functionality (interfaces to host). However, this would impose restrictions on the design decisions of the CSP.

In [34] an initial bitstream is not required. The authors leverage the built-in PUF, ECC and AES cores on SmartFusion-2 FPGAs of Microsemi. The PUF is used to generate a pair of ECC keys, and the client uses the ECC public key to encrypt the AES secret key. The AES key is used to encrypt client bitstream. Partial reconfiguration of the FPGA fabric is not available in these FPGAs, therefore the client should design and implement the interfaces to host, which is inconvenient for the CSP. On the other hand, in an untrusted environment, the FPGA can be run in factory test mode to read out client's IP design.

In [33], [34] the public key is published via standard public key infrastructure held by the trusted party.

These schemes mainly assume temporal multi-tenant setting. Nevertheless, [33], [35] would theoretically work in the spatial setting, however, a malicious tenant can overwrite the configuration of another tenant, since the CSP has neither control over the bitstream generation nor access to plain-text bitstreams. Moreover, the CSP has no guarantees that the encrypted bitstream is harmless, i.e., free of power viruses or sensors. To sum up, for trusted computing on cloud FPGAs, FPGA vendor support is required for IP protection. At the same time, the CSP must be assured that encrypted clients bitstreams contain no harmful logic. We discuss this in more detail in § 9.1.

8.3.2. Access Control & Encrypted Communications.
Access Control. To mitigate unauthorized access to co-clients data and IP in spatial multi-tenant setting, a DMA engine can be implemented in the FPGA shell to allow valid memory accesses only and to isolate the data of the different virtual FPGAs [6], [7], since IOMMU matches memory accesses per physical FPGA only. Whereas in case the FPGA is accessed through the network, the network interface controller implemented in the FPGA shell is augmented with VLAN-tagging to isolate the packets of different virtual FPGAs [9]. These solutions work as the first line of defense. Hence, for further protection of data at rest and in transit, encryption is used.

Elnaggar et al. [98] proposed to add an external hardware component to the FPGA, rather than the FPGA shell, secure authentication module (SAM) that manages the distribution of secret keys to clients and enforces access control to the FPGA-accelerated tasks based on these keys. A client then integrates the secret key in the SW-application and the FPGA bitstream and uploads them to the cloud. At runtime, SAM verifies that the SW-application and the requested FPGA-accelerated task belong to the same legitimate client by comparing the implanted secret key. If authenticated, the SAM sends both components a secret session key to encrypt their communications. Hence, the established trust relies on the security of the secret key given to the client and implanted in the bitstream. Therefore, a client bitstream must be protected, otherwise, the security of the entire scheme is compromised.

Encrypted Communications. Yazdanshenas et al. [99], [100] attempt to address the confidentiality of off-chip communication using encryption in the FPGA shell, the statically configured logic, for all incoming/outgoing traffic of a physical FPGA through its interfaces (PCIe, Network and off-chip memory storage). Another way to secure tenant communications is to add encryption core wrappers around the reconfigurable regions to encrypt the data before leaving the virtual FPGA. This prevent I/O bottleneck compared to the first option, where all tenants' data gets encrypted or decrypted in the shell. However, the choice between the two modes should consider the trust assumptions: opting for shell encryption implies trust in the shell and the CSP, whereas using an encryption wrapper per virtual FPGA protects the virtual FPGA traffic in an untrusted environment. The main issue with the latter mode is that clients must securely bring their own keys into the encryption wrappers, which we only see feasible if bitstream protection on cloud FPGAs is supported. Nevertheless, for both encryption modes, defenses against remote physical attacks should be in-place. The authors compared the cost of soft vs. built-in AES cores and recommended the integration of built-in cryptographic cores by the FPGA vendors. As an example, the authors proposed to leverage a hardwired Network-on-Chip (NoC) where the NoC can be used to connect these regions via encryption-enhanced routers [99], [100]. However, this would result in extensive architectural modifications to FPGAs and restrict the CSP choices, i.e., number and size of virtual FPGAs.

In summary, in temporal or spatial multi-tenant settings, data isolation or encryption of different tenants is performed either i) by the client logic implying client bitstream protection, or ii) by the FPGA shell implying trust in the shell and the CSP. This trust can be simply assumed as a perquisite or gained, e.g. by remote attestation. More discussions on this follow in § 9.1.

9. Discussion

In light of the analysis of the security concerns that would arise from the different cloud FPGA computing deployment models, we present next our insights on the open challenges and opportunities in establishing trust in FPGA-based computing in the cloud in § 9.1 and lessons learned from trusted computing on CPUs in § 9.2.

9.1. Trusted Computing on Cloud FPGAs

FPGA devices can be leveraged to establish the trust anchor required in different secure computation flows and services on the cloud, owing to their features and advantages to both software as well as hardwired static hardware. One prominent example is offering to run clients' workloads in a trusted execution environment that gets established in cloud FPGAs. This is analogous to running workloads in a trusted execution environment on a CPU in the cloud, which is being offered more by CSPs recently to provide confidentiality and integrity guarantees both in bare-metal instances [55], [56] and in virtual machines [54]. We envision the FPGA shell, or hypervisor, to set up the FPGA trusted execution environment or trust anchor, i.e., to configure the client IP bitstream through the internal configuration port on the FPGA. Moreover, the FPGA shell must ensure that the bitstream is actually configuring the intended/allocated reconfigurable region (virtual FPGA) and not overwriting the configuration of another virtual FPGA, if spatial multi-tenancy is enabled.

One significant open challenge here is establishing a trust anchor when the FPGA shell, also a bitstream itself, is untrusted. Therefore, a supporting infrastructure (architecturally in FPGA devices as well as within overlying protocols and toolchains) is required to establish trust in the FPGA shell in the first place. Then, it can provide clients with sufficient guarantees that their IPs and data remain contained in secure and untampered isolation during execution and at rest.

FPGA-TCB. We refer to the components on the FPGA that are required to establish a trust anchor as the FPGA Trusted Computing Base (FPGA-TCB). This TCB would be required to verify the integrity of the FPGA shell and configure it on the FPGA, similar to secure boot on CPUs. The FPGA-TCB should also provide remote attestation of the FPGA shell, or the part of it that establishes and manages the FPGA trusted execution environment. Remote attestation would be required to provide assurance to the clients regarding the integrity of the deployed cryptographic primitives used to decrypt and verify their IP bitstreams and the protected access to plain-text bitstreams. To enable FPGA shell remote attestation and client bitstream decryption, the FPGA-TCB would also be required to handle secret session keys generation, exchange and management. Note that some FPGA vendors offer some of these requirements in their recent FPGAs, e.g., Intel Black Key Provisioning enables remote and secure provisioning of the AES root key in Stratix 10 [101].

Since the immutable FPGA-TCB cannot be compromised by malicious parties, it forms a root of trust in the FPGA. By leveraging remote attestation, the trust is extended to the FPGA shell, specifically to FPGA shell components that deal with client IPs and data, e.g., decryption and configuration management logic as well as the logic that prevents plain-text bitstream readback through the different configuration ports.[5] In [102] a first step towards the establishment of trust anchor on cloud FPGA is made.

This calls for a comprehensive analysis of the *minimal* FPGA-TCB components (both hardware and software) to provide these required security guarantees. This includes investigating efficient secure key and certificate provi-

5. Readback property enables reading configuration memory content (LUTs and programmable connections) to support the implementation of error correction for reliability.

sioning as well as the minimal hardware primitives (e.g., cryptographic processors or cores and their verified side-channel-resilient implementations). Furthermore, which of these primitives must be hard-wired into the FPGA, and which can be provided as configurable logic? How much hardware-software co-design is needed and what are the ideal design choices therein? How can debug interfaces, such as the Joint Test Action Group (JTAG) interface among others, be provided securely to still enable cloud clients to debug their FPGA development without threatening the confidentiality of other IPs and computations running on the FPGAs? Last but not least, how can we provide bitstream validation while still preserving client IP protection where the bitstream is encrypted? Various usability/performance/security trade-offs are involved within these open questions, all of which require a comprehensive and systematic treatment and analysis.

Remote Physical Attacks & Defenses. On another front, remote physical attacks demonstrate and emphasize why *secure* spatial multi-tenant FPGA computing is challenging. Nevertheless, in quest for maximum utilization and as FPGA devices' capacities keep scaling, spatial multi-tenancy is becoming inevitable. Thus, it becomes imperative to develop mechanisms that provide the required future-proof security assurances that are unique to these emerging FPGA computing scenarios. Runtime defenses, presented in § 8.2, require the implementation of protection logic in the FPGA shell or in a client's allocated reconfigurable region to protect the victim logic against a specific leakage source, e.g., power leakage. Such defenses do not only consume non-negligible resources but also require the clients to have FPGA hardware design expertise in order to implement them and find the best trade-off between security and resources overhead. Whereas other runtime defenses require the implementation of configurable sensors and noise generators, which are considered malicious and would be prevented by a bitstream validation process.

Fundamentally, the key enabler of these attacks is that clients can freely configure their (malicious) logic on the FPGA fabric. Developing mechanisms for effectively scanning and validating the clients' logic for malicious circuit signatures seems to be the intuitive approach to prevent malicious logic from being configured on the FPGA in the first place. While recently proposed stand-alone tools for detection of malicious circuits and suspicious behavior show promising results, they detect known malicious circuits only, but provide no guarantees for future zero-day attacks. Moreover, these tools, deployed by the CSP, work at the netlist level and require reverse engineering of clients' bitstreams. This is especially a challenge when the bitstreams are encrypted for purposes of IP protection, as pointed out above.

Therefore, FPGA toolchains may need to additionally cater for providing such security guarantees in cloud deployment settings, e.g., by extending them with particular design rule checks to enforce certain constraints on routing (e.g., to eliminate potential crosstalk across co-located tenants) and checking for potentially malicious design primitives before generating the bitstream.

9.2. Trusted Computing on CPUs: Takeaways

Being the more established and classical computing setting in the cloud, various lessons can be drawn from CPU-based computing. These can be taken into account in proactively assessing the security challenges and requirements of FPGA-based computing in the cloud, as well as adapting defenses that can also apply for FPGAs and drawing analogies where possible. In the following, we discuss the most common examples of security threats in CPU-based computing and draw the analogy to FPGAs.

Software Runtime Attacks. Analogously to TEE infrastructures and SDKs in CPU-based computing, trust assumptions in FPGA SDKs are to be examined and questioned. The critical security implications of a memory corruption vulnerability, if found in the SGX SDK, have been demonstrated in [103], where the authors show how a vulnerability can be exploited to craft a return-oriented-programming attack chain in the Trusted Runtime System (tRTS) code handling enclave entry. Similarly, such vulnerabilities can be exploited to load unintended accelerator tasks on the FPGA or even inject malicious accelerator tasks [98]. Therefore, SDKs and toolchains for FPGA development, whether at the client's end or hosted by the CSP in the cloud, need to be scrutinized and tested for such vulnerabilities and hardened accordingly, such that trust assumptions regarding them are sufficiently justified.

Microarchitectural Attacks. The recent outbreak of microarchitectural attacks in CPU-based computing also motivates an analogous investigation in FPGA-based computing. Identical counterparts of attacks such as Spectre and Meltdown [104]–[109] are not feasible on FPGA devices since they exploit side effects of specific microarchitectural features, e.g., speculative execution, that simply do not exist on FPGA fabric. However, attacks that generally exploit shared resources, e.g. caches and other memory buffers (store, load, etc.), would principally be also feasible on FPGA devices in one form or another, if they were shared without isolation among FPGA co-tenants. We already discuss in more detail in § 5 how different types of cache attacks would also be feasible on FPGA devices, and which defenses would similarly apply. In principle, besides caches, all shared microarchitectural/hardware/logic resources by co-tenants on an FPGA device constitute a potential security threat that require a thorough security information flow analysis under the assumed threat model of the deployment settings in question. They may leak information on the execution of one tenant to another via an exploitable side channel, unless appropriate defenses, e.g., flushing, partitioning or randomization, are applied.

Power Drop & Power Leakage Attacks. CPU vendors provide software-accessible interfaces to dynamic voltage and frequency scaling to control the supply voltage and the operating frequency of different CPU components (cores, last-level caches, etc.) at runtime and are accessible only to privileged software. Recent attacks exploit these interfaces to maliciously drop the supply voltage [110], [111] or increase operating frequency [112] to precisely induce faults in the operations of victim applications to reveal confidential data, e.g., secret keys. Some CPU vendors further provide software-accessible interface to power measurements of individual components (cores, uncore, DRAM, etc.). This interface has been exploited in [53] to infer executed instructions and their operands in a victim application to extract secret data. Similarly, such attacks have already been shown feasible on FPGAs, although by unprivileged malicious co-tenants in § 7 and § 6. Nevertheless, in an untrusted environment, such attacks can also be launched by the privileged FPGA shell that comprises power viruses or sensors. Defenses mitigating these attacks on CPUs usually involve patches to disable the interfaces, which is not directly applicable for FPGA devices, since the attacks stem from configuring malicious logic on the FPGA fabric. To mitigate these attacks on FPGA devices, both the FPGA shell bitstream as well as the clients' bitstreams should be validated and vetted for malicious logic before configuration.

In summary, valuable lessons can be drawn from CPU-based computing for FPGA-based computing in the cloud; a rigorous and systematic security analysis of the underlying fabric and TCB early on is crucial for the derived security guarantees for all that lies on top of that. Furthermore, for FPGAs, unlike CPUs, the *configurable* fabric is a double-edged sword. On one hand, it allows seamless updating and patching of hardware and vulnerabilities therein. However, it also poses an even larger attack surface if exposed to malicious parties to configure with complete freedom. Therefore, providing techniques to vet and police configuration bitstreams while preserving their confidentiality is the key open and non-trivial challenge for FPGA-based trusted computing.

10. Conclusion

In this paper, we systematically visited the different deployment models of FPGA-based cloud computing, and highlighted their adversary models and required security guarantees. The threat landscape is diverse and spans different levels of the deployment stack. We categorized and analyzed the different feasible attacks in FPGA-based cloud computing with a focus on remote physical attacks. We further classified state-of-the-art countermeasures against remote physical attacks into preventive and runtime defenses and discussed their deployment and security guarantees. Bitstream validation, a preventive mechanism, is more comprehensive and effective against the different threats, whereas current runtime defenses are each tailored to detect or mitigate individual threats.

We concluded with our insight into the open challenges in establishing trust in FPGA-based computing in the cloud and identifying the required minimal trusted computing base on FPGA devices required to provide clients with the relevant security and IP assurance guarantees. We also discuss some takeaways and lessons learned from CPU-based trusted computing and the security challenges therein, and draw potential analogies for FPGA-based computing. We emphasize that developing mechanisms to vet and validate protected (e.g. encrypted) configuration bitstreams is the key open and non-trivial challenge for FPGA-based trusted computing.

Acknowledgment. We thank the anonymous reviewers for their valuable and constructive feedback. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 CROSSING/236615297 and the Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS).

References

[1] M. Research, "Project Catapult," https://www.microsoft.com/en-us/research/project/project-catapult.

[2] Microsoft Research, "Project Brainwave," https://www.microsoft.com/en-us/research/project/project-brainwave.

[3] Amazon AWS, "Amazon EC2 F1," https://aws.amazon.com/ec2/instance-types/f1.

[4] Y. Xu, O. Muller, P.-H. Horrein, and F. Pétrot, "Hcm: an abstraction layer for seamless programming of DPR FPGA," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012.

[5] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: booting virtualized hardware accelerators with openstack," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2014.

[6] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang, "Enabling FPGAs in the cloud," in *ACM Conference on Computing Frontiers*. ACM, 2014.

[7] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2015.

[8] J. Weerasinghe, F. Abel, C. Hagleitner, and A. Herkersdorf, "Enabling FPGAs in hyperscale data centers," in *IEEE Intl Conf on Ubiquitous Intelligence and Computing and IEEE Intl Conf on Autonomic and Trusted Computing and IEEE Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*. IEEE, 2015.

[9] O. Knodel, P. Lehmann, and R. G. Spallek, "RC3E: reconfigurable accelerators in data centres and their provision by adapted service models," in *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2016.

[10] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, protection, and compatibility for reconfigurable fabric with amorphos," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2018.

[11] D. R. Gnad, F. Oboril, and M. B. Tahoori, "Voltage drop-based fault attacks on FPGAs using valid bitstreams," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017.

[12] K. Matas, T. La, N. Grunchevski, K. Pham, and D. Koch, "Invited tutorial: FPGA hardware security for datacenters and beyond," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2020.

[13] J. Krautter, D. R. Gnad, and M. B. Tahoori, "FPGAhammer: remote voltage fault attacks on shared FPGAs, suitable for DFA on AES," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018.

[14] D. Mahmoud and M. Stojilović, "Timing violation induced faults in multi-tenant FPGAs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019.

[15] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "An inside job: remote power analysis attacks on FPGAs," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018.

[16] J. Gravellier, J.-M. Dutertre, Y. Teglia, and P. Loubet-Moundi, "High-speed ring oscillator based sensors for remote side-channel attacks on fpgas," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2019.

[17] O. Glamočanin, L. Coulon, F. Regazzoni, and M. Stojilović, "Are cloud FPGAs really vulnerable to power analysis attacks?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[18] M. Zhao and G. E. Suh, "FPGA-based remote power side-channel attacks," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018.

[19] J. Gravellier, J.-M. Dutertre, Y. Teglia, P. L. Moundi, and F. Olivier, "Remote side-channel attacks on heterogeneous SoC," in *Smart Card Research and Advanced Applications*, S. Belaïd and T. Güneysu, Eds. Springer International Publishing, 2020.

[20] J. Krautter, D. Gnad, and M. Tahoori, "CPAmap: on the complexity of secure FPGA virtualization, multi-tenancy, and physical design," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020.

[21] I. Giechaskiel, K. Rasmussen, and J. Szefer, "Reading between the dies: cross-SLR covert channels on multi-tenant cloud FPGAs," in *IEEE International Conference on Computer Design (ICCD)*. IEEE, 2019.

[22] D. R. Gnad, C. D. K. Nguyen, S. H. Gillani, and M. B. Tahoori, "Voltage-based covert channels in multi-tenant FPGAs," *IACR Cryptology ePrint Archive*, vol. 2019, p. 1394, 2019.

[23] C. Ramesh, S. B. Patil, S. N. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier, "FPGA side channel attacks without physical access," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018.

[24] G. Provelengios, C. Ramesh, S. B. Patil, K. Eguro, R. Tessier, and D. Holcomb, "Characterization of long wire data leakage in deep submicron FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019.

[25] I. Giechaskiel, K. B. Rasmussen, and K. Eguro, "Leaky wires: information leakage and covert communication between FPGA long wires," in *Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 2018.

[26] I. Giechaskiel, K. Eguro, and K. B. Rasmussen, "Leakier wires: exploiting FPGA long wires for covert-and side-channel attacks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2019.

[27] I. Giechaskiel, K. B. Rasmussen, and J. Szefer, "Measuring long wire leakage with ring oscillators in cloud FPGAs," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019.

[28] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "Remote inter-chip power analysis side-channel attacks at board-level," in *International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, 2018.

[29] I. Giechaskiel, K. Rasmussen, and J. Szefer, "C3APSULe: cross-FPGA covert-channel attacks through power supply unit leakage," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.

[30] S. Tian and J. Szefer, "Temporal thermal covert channels in cloud FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2019.

[31] Intel, "Disaggregated servers drive data center efficiency and innovation," https://www.intel.com/content/www/us/en/it-management/intel-it-best-practices/disaggregated-server-architecture-drives-data-center-efficiency-paper.html.

[32] S. Biookaghazadeh, M. Zhao, and F. Ren, "Are FPGAs suitable for edge computing?" in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge)*. USENIX Association, 2018.

[33] K. Eguro and R. Venkatesan, "FPGAs for trusted cloud computing," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2012.

[34] B. Hong, H.-Y. Kim, M. Kim, T. Suh, L. Xu, and W. Shi, "Fasten: an FPGA-based secure system for big data processing," *IEEE Design & Test*, 2017.

[35] M. E. Elrabaa, M. Al-Asli, and M. Abu-Amara, "Secure computing enclaves using FPGAs," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 2019.

[36] I. Kuon, R. Tessier, and J. Rose, *FPGA Architecture: Survey and Challenges*. now, 2008.

[37] D. Koch, *Partial reconfiguration on FPGAs: architectures, tools and applications*. Springer Science & Business Media, 2012, vol. 153.

[38] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *IEEE International Symposium on Computer Architecture (ISCA)*. IEEE, 2014.

[39] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur *et al.*, "Configurable clouds," *IEEE Micro*, 2017.

[40] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: SmartNICs in the public cloud," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2018.

[41] H. Cloud, "FPGA accelerated cloud server," https://www.huaweicloud.com/en-us/product/facs.html.

[42] A. Cloud, "Fpga-based compute-optimized instance families," https://www.alibabacloud.com/help/doc-detail/108504.htm, 2019.

[43] Baidu, "Baidu cloud compute," https://cloud.baidu.com/product/fpga.html.

[44] Telekom, "Telekom ECS," https://open-telekom-cloud.com/en/products-services/elastic-cloud-server.

[45] C. Plessl and M. Platzner, "Virtualization of hardware-introduction and survey." in *ERSA*, 2004.

[46] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on FPGA virtualization," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.

[47] D. L. How and S. Atsatt, "Sectors: divide conquer and softwarization in the design and validation of the Stratix® 10 FPGA," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016.

[48] Z. István, G. Alonso, and A. Singla, "Providing multi-tenant services with FPGAs: case study on a key-value store," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.

[49] R. Kenny, "Soc FPGA hardware security requirements and roadmap," *Intel SoC FPGA Development Forum (ISDF16)*, 2016.

[50] P. Koeberl, "Multi-tenant FPGA security: challenges and opportunities," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2020.

[51] T. Trippel, K. G. Shin, K. B. Bush, and M. Hicks, "An extensible framework for quantifying the coverage of defenses against untrusted foundries," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.

[52] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, 2014.

[53] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss, "PLATYPUS: software-based power side-channel attacks on x86," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021.

[54] Microsoft, "DCsv2-series VM now generally available from azure confidential computing," https://azure.microsoft.com/en-us/blog/dcsv2series-vm-now-generally-available-from-azure-confidential-computing.

[55] IBM, "Data-in-use protection on IBM cloud using Intel SGX," https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[56] Alibaba, "ECS bare metal instance," https://www.alibabacloud.com/product/ebm.

[57] J. Krautter, D. R. Gnad, and M. B. Tahoori, "Mitigating electrical-level attacks towards secure multi-tenant FPGAs in the cloud," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2019.

[58] T. La, K. Mätas, N. Grunchevski, K. Pham, and D. Koch, "FPGADefender: malicious self-oscillator scanning for Xilinx UltraScale+ FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2020.

[59] S. Trimberger and S. McNeil, "Security of FPGAs in data centers," in *International Verification and Security Workshop (IVSW)*. IEEE, 2017.

[60] Y. Luo and X. Xu, "Hill: a hardware isolation framework against information leakage on multi-tenant FPGA long-wires," in *IEEE International Conference on Field-Programmable Technology (FPT)*. IEEE, 2019.

[61] Z. Seifoori, S. S. Mirzargar, and M. Stojilović, "Closing leaks: routing against crosstalk side-channel attacks," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2020.

[62] J. Krautter, D. R. Gnad, F. Schellenberg, A. Moradi, and M. B. Tahoori, "Active fences against voltage-based side channels in multi-tenant FPGAs," in *International Conference On Computer Aided Design (ICCAD)*. IEEE/ACM, 2019.

[63] M. Nassar, S. Bhasin, J.-L. Danger, G. Duc, and S. Guilley, "BCDL: a high speed balanced dpl for FPGA with global precharge and no early evaluation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2010.

[64] A. Wild, A. Moradi, and T. Güneysu, "Glifred: glitch-free duplication towards power-equalized circuits on FPGAs," *IEEE Transactions on Computers*, 2017.

[65] Z. Chen, S. Haider, and P. Schaumont, "Side-channel leakage in masked circuits caused by higher-order circuit effects," in *International Conference on Information Security and Assurance (ISA)*. Springer, 2009.

[66] K. H. Boey, Y. Lu, M. O'Neill, and R. Woods, "Random clock against differential power analysis," in *IEEE Asia Pacific Conference on Circuits and Systems*. IEEE, 2010.

[67] T. Güneysu and A. Moradi, "Generic side-channel countermeasures for reconfigurable devices," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2011.

[68] Y. Lu, M. P. O'Neill, and J. V. McCanny, "FPGA implementation and analysis of random delay insertion countermeasure against dpa," in *IEEE International Conference on Field-Programmable Technology (FPT)*. IEEE, 2008.

[69] A. Moradi and O. Mischke, "On the simplicity of converting leakages from multivariate to univariate," in *International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2013.

[70] D. Jayasinghe, A. Ignjatovic, and S. Parameswaran, "Rftc: runtime frequency tuning countermeasure using FPGA dynamic reconfiguration to mitigate power analysis attacks," in *Design Automation Conference (DAC)*. ACM, 2019.

[71] G. Provelengios, D. Holcomb, and R. Tessier, "Characterizing power distribution attacks in multi-user FPGA environments," in *IEEE International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019.

[72] *Security Monitor Product Brief*, Xilinx Inc., 2019.

[73] *Intel FPGA Temperature Sensor IP Core User Guide*, Intel Corporation, 05 2018.

[74] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: efficient rowhammer on heterogeneous FPGA-CPU platforms," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020.

[75] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level Protection Against Cache-based Side Channel Attacks in the Cloud," in *USENIX Security Symposium*. USENIX Association, 2012.

[76] Z. Zhou, M. K. Reiter, and Y. Zhang, "A Software Approach to Defeating Side Channels in Last-Level Caches," in *ACM Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[77] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[78] Z. Wang and R. B. Lee, "New Cache Designs for Thwarting Software Cache-based Side Channel Attacks," in *IEEE International Symposium on Computer Architecture (ISCA)*. ACM, 2007.

[79] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection," in *Design Automation Conference (DAC)*. ACM, 2016.

[80] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory," in *USENIX Security Symposium*. USENIX Association, 2017.

[81] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," in *IEEE Micro*, 2018.

[82] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments," in *USENIX Security Symposium*. USENIX Association, 2020.

[83] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems," in *Design Automation Conference (DAC)*. ACM, 2018.

[84] M. K. Qureshi, "Ceaser: Mitigating Conflict-based Cache Attacks via Encrypted-Address and Remapping," in *IEEE Micro*, 2018.

[85] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security Symposium*, 2019.

[86] Q. K. Zhu, *Power distribution network design for VLSI*. John Wiley & Sons, 2004.

[87] *7 series FPGAs PCB design guide*, Xilinx Inc., 5 2019, rev. 1.14.

[88] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Moats and drawbridges: an isolation primitive for reconfigurable hardware based systems," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2007.

[89] *UltraScale architecture and product data sheet: overview*, Xilinx Inc., 8 2019, rev. 3.10.

[90] G. Piret and J.-J. Quisquater, "A differential fault attack technique against SPN structures, with application to the AES and KHAZAD," in *International workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2003.

[91] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, 2011.

[92] S. Tian, W. Xiong, I. Giechaskiel, K. Rasmussen, and J. Szefer, "Fingerprinting cloud FPGA infrastructures," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ACM, 2020.

[93] K. M. Zick, M. Srivastav, W. Zhang, and M. French, "Sensing nanosecond-scale voltage attacks and natural transients in FPGAs," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2013.

[94] T. Sugawara, K. Sakiyama, S. Nashimoto, D. Suzuki, and T. Nagatsuka, "Oscillator without a combinatorial loop and its threat to FPGA in data centre," *Electronics Letters*, 2019.

[95] A. Duncan, A. Nahiyan, F. Rahman, G. Skipper, M. Swany, A. Lukefahr, F. Farahmandi, and M. Tehranipoor, "SeRFI: secure remote FPGA initialization in an untrusted environment," in *VLSI Test Symposium (VTS)*. IEEE, 2020.

[96] M. Hoffmann and C. Paar, "Stealthy opaque predicates in hardware-obfuscating constant expressions at negligible overhead," *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2018.

[97] J. Kataria, R. Housley, J. Pantoga, and A. Cui, "Defeating cisco trust anchor: A case-study of recent advancements in direct FPGA bitstream manipulation," in *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2019.

[98] R. Elnaggar, R. Karri, and K. Chakrabarty, "Multi-tenant FPGA-based reconfigurable systems: attacks and defenses," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019.

[99] S. Yazdanshenas and V. Betz, “Improving confidentiality in virtualized FPGAs,” in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2018.

[100] S. Yazdanshenas and V. Betz, “The costs of confidentiality in virtualized FPGAs,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[101] *Intel Stratix 10 Device Security User Guide*, Intel Corporation, 10 2020.

[102] S. Zeitouni, J. Vliegen, T. Frassetto, D. Koch, A.-R. Sadeghi, and N. Mentens, “Trusted configuration in cloud FPGAs,” in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), to be published.* IEEE, 2021.

[103] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, “The guard’s dilemma: efficient code-reuse attacks against intel SGX,” in *USENIX Security Symposium.* USENIX Association, 2018.

[104] P. Kocher, J. Horn, A. Fogh, D. Genkin *et al.*, “Spectre attacks: Exploiting speculative execution,” in *IEEE Symposium on Security and Privacy (S&P).* IEEE, 2019.

[105] C. Canella, D. Genkin, L. Giner, D. Gruss *et al.*, “Fallout: Leaking data on meltdown-resistant cpus,” in *ACM Conference on Computer and Communications Security (CCS).* ACM, 2019.

[106] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “Zombieload: Cross-privilege-boundary data sampling,” in *ACM Conference on Computer and Communications Security (CCS).* ACM, 2019.

[107] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “Ridl: Rogue in-flight data load,” in *IEEE Symposium on Security and Privacy (S&P).* IEEE, 2019.

[108] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, “SGAxe: how SGX fails in practice,” https://sgaxeattack.com, 2020.

[109] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution,” in *IEEE Symposium on Security and Privacy (S&P).* IEEE, 2019.

[110] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, “V0LTpwn: attacking x86 processor integrity from software,” in *USENIX Security Symposium.* USENIX Association, 2020.

[111] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, “Plundervolt: software-based fault injection attacks against intel SGX,” in *IEEE Symposium on Security and Privacy (S&P).* IEEE, 2020.

[112] A. Tang, S. Sethumadhavan, and S. Stolfo, “CLKSCREW: exposing the perils of security-oblivious energy management,” in *USENIX Security Symposium.* USENIX Association, 2017.

[113] J. S. Wong, P. Sedcole, and P. Y. Cheung, “Self-measurement of combinatorial circuit delays in FPGAs,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2009.

[114] K. M. Zick and J. P. Hayes, “Low-cost sensing with ring oscillator arrays for healthier reconfigurable systems,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2012.

[115] D. R. Gnad, F. Oboril, S. Kiamehr, and M. B. Tahoori, “Analysis of transient voltage fluctuations in FPGAs,” in *IEEE International Conference on Field-Programmable Technology (FPT).* IEEE, 2016.

Appendix A. Hardware Circuits for Remote Physical Attacks

A.1. Reconfigurable Sensors

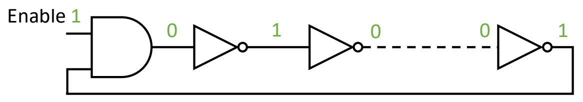
While on-die sensors to monitor FPGA supply voltage and temperature are typically available on recent generation of FPGAs, their response time and sampling frequency are typically not sufficient to capture short time-constant effects such as voltage transients. Therefore, fine-grain sensors that can measure the impact of physical properties (e.g., voltage and temperature variations in power distribution network, aging, etc.) of the different regions on the reconfigurable logic have been proposed [113], [114]. Two prominent examples of configurable sensors, which are implemented in the FPGA fabric that can perform self-measurement of their path delays are the ring oscillator and delay-line based sensors. Since propagation delay in circuits is highly affected by process, voltage and temperature (PVT) variations, measurement of combinatorial [6] propagation delays provides a means of measuring fine-grained changes in operating conditions and process variations. Nevertheless, calibration or post-processing steps might be required to suppress the undesired effects [114], [115].

Ring Oscillator (RO) Sensor consists of an AND gate (enabler) and an odd number of serially-chained inverters, referred to as stages, where the output of the last stage is fed back to the input of the AND gate (along with an enable signal), thus forming a *combinatorial loop* circuit. Further, the AND gate and one inverter may be combined in NAND gate. Other *sequential* RO sensors are the latch-based RO and the flip-flop based RO [94]. When a RO circuit is enabled, the RO output signal alternates between ‘1’ and ‘0’ continuously. To measure RO *oscillation frequency*, the RO output is used to trigger a counter circuit. The counter value is sampled at a user-defined rate, such that changes in the counter value between successive samples of the same RO reflect changes in operating conditions. RO sensors used in the different remote physical attacks differ in the number of stages, the counter design (binary or non-binary) and whether sequential elements (latches or flip-flops) are used. A classical ring oscillator (RO) is shown in Fig. 4a. All inverters but one can also be replaced with digital buffers as shown in Fig. 4b. Both RO designs Fig. 4a and Fig. 4b are *pure combinatorial* logic circuits, whereas the latch-based RO and the flip-flop (FF) based RO, shown in Fig. 4c and Fig. 4d respectively, are sequential. RO/oscillation frequency is inversely proportional to the delay of a signal traversing the feedback loop through the AND gate and chained inverters/buffers. Therefore, the oscillation frequency is dependent on the number of inverters/buffers and their propagation delays.

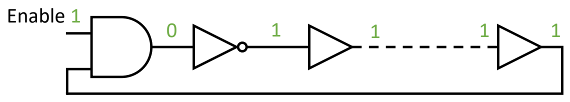
Delay-Line Sensor consists of a chain of n buffers, through which a clock signal propagates. Propagation delay is measured by how far the clock propagates through the buffers within a fixed time frame using a time-to-digital convertor (TDC) circuit. Latches are tapping into the connected buffers such that the propagation delay in the delay-line is measured by observing the boundary between all ‘1’s latches and all ‘0’s latches. The n latches are sampled at some user-defined rate and the obtained value is encoded into a $log_2(n)$-bit binary value, such that changes in the binary value between successive samples reflect changes in operating conditions. To minimize its area overhead (latches and encoder complexity), delay-line buffers can be split into initial and observable, such that only the observable buffers are connected to the latches.

Zhao and Suh [18] examined the trade-off between RO and delay-line based sensors in terms of sampling

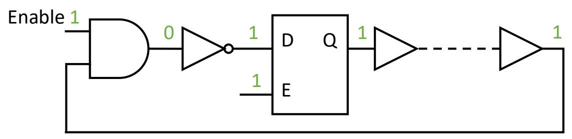
6. Combinatorial circuits consist only of wires and logic gates i.e. no memory elements such as flip-flops.



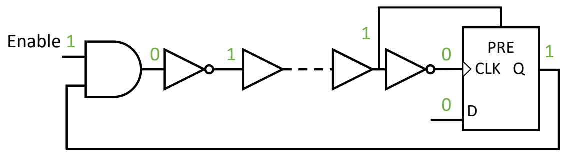
(a) Pure combinatorial RO design of odd number of inverters/stages.



(b) Pure combinatorial RO design of one inverter & even number of buffers.



(c) Sequential RO design with a D-latch [27].



(d) Sequential RO design with a D-FF [27].

Figure 4. Different RO Designs.

frequency, accuracy, resolution and complexity (area overhead and ease of implementation). In general, determining which on-chip sensor to deploy and its sampling frequency highly depends on the physical properties to be measured.

A.2. Reconfigurable Power Viruses

Examples of configurable circuits that have been used for power dissipating in the literature are: RO-based, FF-based and PIP-based power viruses. By toggling a large group of such power viruses for enough time, voltage drops (both transient and steady-state drops) can be induced in the power distribution network of the FPGA. **FF-based** power virus consists of a D flip-flop (D-FF), whose output is connected to the input of an inverter, while the inverter output is fed back to the input of the D-FF [115]. **PIP-based** power virus is constructed by leveraging programmable interconnect points (PIPs) in Xilinx FPGAs. These are programmable transistors used to connect inputs and outputs of I/O blocks and CLBs into the routing network in the FPGA fabric. Unused PIPs and wires in the FPGA fabric resemble a high capacitance, such that frequently charging and discharging them result in transient voltage drop and overshoot in the PDN [93].

An on-chip power dissipating circuit can be precisely controlled to induce power glitches within a specific time-frame. Moreover, such power viruses can be activated for long enough time to heat the FPGA. Therefore, power dissipating circuits can be leveraged to mount power fault attacks, as well as on-chip noise generators for mitigating side-channel attacks, as we describe later in more detail.

Appendix B. Details on Attacks Exploiting Crosstalk

Attack [23]. The key of the final round of the AES-128 can be recovered byte by byte. Then the encryption key can be obtained by inverting the key schedule. This requires the attacker and victim circuits to have 16 transmitter-receiver pairs of neighboring long wires, with one pair for each byte of the final round key in case of a high-throughput AES implementation with a datapath of 128-bit. Consequently, this is reduced to one pair of neighboring long wires if the AES has an 8-bit datapath.

The crosstalk effect increases for longer transmission pair and lower operating frequencies of the victim logic, which directly influences the number of encryptions required to recover the key. A minimum of 217 encryptions are required to recover the key for an AES core running at 10KHz, whereas at a higher operating frequency of 4MHz a 1.5 million encryptions are required. Interestingly, this crosstalk effect can also be leveraged to build an adjacency map for long wires within each channel in the FPGA [24], which is considered a proprietary information.

Attacks [25] and [26]. The crosstalk effect is demonstrated in the context of integrating encrypted or obfuscated third-party FPGA IP designs from multiple vendors in the same design [25]. The propagation delay of the receiver's long wire is highly influenced by how long a neighboring victim long wire carries a signal '1' within a sampling period. Thus, the counter value, i.e., the frequency of the RO connected to the receiver, indicates the hamming weight of the transmitted bits over the victim long wire within that sampling period. A sliding-window scheme is proposed to sample the counter at overlapping periods to extract the value of each transmitted bit of a secret key. By subtracting two overlapped hamming weights, information on the first and last bits of the corresponding windows are revealed. Assuming that the victim long wire is transmitting the whole key bits sequentially, and using a sliding-window of size w bits and a secret key of size N bits, where N is a multiple of w, i.e., $N = nw$, the authors compute the probability of recovering the key:

$$P = \left(1 - \frac{1}{2^{n-1}}\right)^{w} \tag{1}$$

This implies that the probability of key recovering is higher for larger keys (asymmetric vs. symmetric keys) and smaller window sizes. Equation 1, however, does not take into account noisy hamming weight measurements due to operating conditions (transmission frequency) and layout of transmission pair (length of transmission lines and distance between them). The sliding-window approach is applied to 32-bit keys in [26]. The authors demonstrate that using a sliding window of size $w = 4$ bits, 98.4% of the 32-bit key can be recovered assuming a single bit stays constant on the transmitter for 1.28 μs, i.e., 780 KHz signal frequency, and a transmission pair of at least 1-segment long wire. Moreover, it requires connecting four counters (equal to the window size) to the snooping circuit, where these counters are sampled at overlapping periods of 5.12 μs (4 $\times$ 1.28 μs).

I TRUSTED CONFIGURATION IN CLOUD FPGAS

[9] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, Nele Mentens. “Trusted Configuration in Cloud FPGAs”. In Proceedings of the 29th IEEE International Symposium On Field-Programmable Custom Computing Machines (FCCM’21), 2021.

2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 978-1-6654-3555-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/FCCM51124.2021.00036

Trusted Configuration in Cloud FPGAs

Shaza Zeitouni*, Jo Vliegen†, Tommaso Frassetto*, Dirk Koch‡, Ahmad-Reza Sadeghi* and Nele Mentens†§
*Technische Universität Darmstadt, Germany, {shaza.zeitouni, tommaso.frassetto, ahmad.sadeghi}@trust.tu-darmstadt.de
†Katholieke Universiteit Leuven, Belgium, {jo.vliegen, nele.mentens}@kuleuven.be
‡The University of Manchester, UK, dirk.koch@manchester.ac.uk
§Leiden University, The Netherlands, n.mentens@liacs.leidenuniv.nl

***Abstract*—In this paper we tackle the open paradoxical challenge of FPGA-accelerated cloud computing: On one hand, clients aim to secure their Intellectual Property (IP) by encrypting their configuration bitstreams prior to uploading them to the cloud. On the other hand, cloud service providers disallow the use of encrypted bitstreams to mitigate rogue configurations from damaging or disabling the FPGA. Instead, cloud providers require a verifiable check on the hardware design that is intended to run on a cloud FPGA at the netlist-level before generating the bitstream and loading it onto the FPGA, therefore, contradicting the IP protection requirement of clients. Currently, there exist no practical solution that can adequately address this challenge.**

We present the first practical solution that, under reasonable trust assumptions, satisfies the IP protection requirement of the client and provides a bitstream sanity check to the cloud provider. Our proof-of-concept implementation uses existing tools and commodity hardware. It is based on a trusted FPGA shell that utilizes less than 1% of the FPGA resources on a Xilinx VCU118 evaluation board, and an Intel SGX machine running the design checks on the client bitstream.

I. Introduction

Field-Programmable Gate Arrays (FPGAs) are popular implementation platforms for performance enhancement and energy saving in cloud computing. In a recent market search report, Grand View Research mentions that the increased adoption of FPGA resources in the cloud is an important driver for the growth of the FPGA market [1]. FPGAs are already used in commercial cloud platforms as in Amazon EC2 F1 [2], Microsoft Azure Catapult [3] and Alibaba Cloud F3 [4]. Examples of applications that benefit from FPGA acceleration in the cloud are artificial intelligence, financial trading and network encryption. The deployment model of cloud FPGAs is the following: when a client requests FPGA computing resources in the cloud, the cloud service provider (CSP) allocates an FPGA instance (from a pool of FPGA instances) to the client for a specific amount of time.

Recent research presents attacks specific to the FPGA hardware on commercially deployed cloud FPGAs. These attacks allow clients to potentially damage FPGAs hosted in the cloud and consequently disable the computing resources of other clients. A malicious client performs such a denial-of-service (DoS) attack by uploading a design circuit that drains an excessive amount of current from the power supply of the FPGA such that the whole platform stops functioning. This can be done, e.g., with ring oscillators [5], [6] or by invoking short circuits [7]. A real-world DoS attack on EC2 F1 instances is demonstrated in [8].

Other categories of attacks – side and covert-channel attacks – have been mostly shown in academic settings, assuming that different clients share different portions of the same FPGA. This allows a malicious client to exploit crosstalk between the wires of an FPGA [9], [10], voltage fluctuations on the shared power distribution network [11]–[13] or thermal information [14]. For example, to perform a side-channel attack, a malicious client uploads a sensor circuit onto the allocated portion of the FPGA with the goal to retrieve secret data processed by another client sharing the same FPGA fabric simultaneously. While no CSP offers the concurrent use of a single FPGA device among clients, multiple computing resources, including FPGAs, typically share a power supply rail, which represents an attack surface [15], [16]. A detailed analysis of these attacks and their defenses is presented in [17].

Therefore, it is important for CSPs to detect the presence of circuits that have sensor or power draining capabilities before they are loaded on the FPGA. We refer to these circuits as *rogue circuits* in the remainder of this paper. The only way to prevent a rogue circuit from being configured onto an FPGA, is to perform a check on the circuit the client intends to upload. In existing commercial platforms, this check is done by the CSP using the FPGA vendor tools that inspect the netlist, e.g., running Xilinx design rule checks (DRCs). Initiatives in academic research propose the use of virus scanners to check FPGA bitstreams [18], [19].

However, any detection mechanism, be it through the existing commercial platforms or through academic virus scanner tools, needs access to the configuration data representing the circuit that the client intends to upload. Consequently, the client is forced to reveal the Intellectual Property (IP) of the hardware circuit to the CSP, which may violate IP protection policy for companies. Clients would rather send encrypted configuration bitstreams to the service provider. However, the use of encrypted bitstreams does not comply with the requirement of the CSP to check the incoming FPGA configurations before they are loaded onto the FPGA.

Contributions. In this work, we propose `TruFPGA`, the first scheme to satisfy the requirements of both, the clients in protecting their IPs and the CSP in guaranteeing that a check has been done on the presence of rogue circuits in the configuration bitstream. In `TruFPGA`, a design check on a client bitstream is executed in a Trusted Execution Environment (TEE) and a proof of execution is provided to the CSP. We further present two options where the TEE resides either on the client side or CSP side and show the trade-offs between the two options. The CSP loads the encrypted bitstream on the FPGA, where it is decrypted on-the-fly with the help of a trusted FPGA shell that we designed. As such, the CSP has no access to the decrypted bitstream, thus solving the paradox of client's IP protection while preventing rogue FPGA configurations. In `TruFPGA`, we tackle the following challenges:

(1) Secret session keys need to be securely established between cloud FPGAs and clients. This is non-trivial in cloud FPGAs because (a) multiple clients with different secret keys should be supported over time, (b) clients are oblivious to the identities of the FPGA instances assigned to them, i.e., CSP's proprietary information, and (c) the CSP, who controls and configures the FPGA, should not have access to the secret keys used in the FPGA.

(2) The client needs the assurance that (a) a read-out of the FPGA's configuration memory, including client's IP, is disallowed, and (b) an unauthorized configuration of client's application bitstream is denied. This is technically challenging, because the CSP controls the FPGA.

(3) The overall solution must be efficient and incurs minimal or no changes to (a) the FPGA architecture, and (b) the cloud infrastructure, e.g., avoid direct communication channel between clients and cloud FPGAs.

We tackle these challenges in `TruFPGA` that comprises a *trusted shell* on the FPGA leveraging physically unclonable functions (PUFs) for key generation and an overarching security *protocol* between the involved parties. Our *proof-of-concept* implementation is demonstrated with existing tools and commodity hardware. Further, `TruFPGA` is generic and the necessary design checks can be performed through vendor toolchains or academic virus scanner tools.

II. Background

In this section we present a brief background on the security components and concepts used in `TruFPGA`.

A. Trusted Execution Environment (TEE)

Ideally, a TEE guarantees that the code and the data running inside the TEE are protected with respect to confidentiality and integrity. Commercial TEEs include Intel SGX [20], AMD SEV [21] and ARM TrustZone [22]. Being a subject of an active research field, several TEE architectures have been proposed, e.g., Sanctum [23], Sanctuary [24], Keystone [25] and Cure [26]. A TEE is an execution environment with its own hardware and software components. Typically, in a TEE, a security-sensitive application, referred to as an *enclave*, runs in isolation of all software on the system including the untrusted operating system (OS) or the hypervisor. A host process, e.g., the OS, sets up the enclave. This means that the enclave's initial binaries may be manipulated. Therefore, the authenticity and integrity of the enclave's initial binaries are verified before execution through remote or local *attestation*. Only then, confidential data can be communicated to the enclave over a secure channel.

B. Remote Attestation

It enables a party/entity to verify the authenticity and integrity of a piece of code or memory on a remote device. A trust anchor on the device computes a digest of the code or memory content, e.g., using a cryptographic hash function and a secret key shared with the verifier. The verifier compares the received digest to a reference value to verify the remote device's status. In the case of a TEE, the platform's secret key is used for attestation.

C. Physically Unclonable Function (PUF)

Silicon PUFs leverage the uncontrollable manufacturing process variation of integrated circuits as a source of entropy to derive a device-specific cryptographic key or a unique identifier. A PUF is stimulated by an input, *challenge*, to produce a *response*, which depends on both the challenge and the innate physical characteristics of the PUF circuit. Therefore, PUF responses are envisioned to be unique and unpredictable. Nevertheless, PUFs have been shown to be prone to software-based modeling attacks [27]. Such attacks require the collection of a large number of challenge-response pairs (CRPs) of a PUF instance to build a mathematical model that emulates the intended PUF behavior. One of the solutions to mitigate such attacks is to obfuscate the output of a PUF using, for example, a cryptographic hash function, such that an attacker has no access to the actual CRPs of a PUF. Note that PUF-based secret keys can be generated on-the-fly, thus eliminating the need for secure non-volatile key storage. PUF technology has been widely adopted for digital fingerprinting and authentication of IoT devices. Moreover, PUFs have already made their way into some FPGA families, e.g., Intel Stratix-10, and Microsemi SmartFusion-2 for the generation of device-specific secret keys.

III. System and Trust Model

In a typical cloud-computing paradigm, different parties are involved. Cloud service providers *CSPs*, such as Microsoft or Amazon, provide different usage models and services and deploy heterogeneous computing platforms. Such platforms involve a conventional CPU-based host that interfaces with co-processors and accelerators (GPUs, ASICs, FPGAs, etc.), which are in turn supplied by different hardware vendors. FPGAs and FPGA design tools are provided by *FPGA vendors*. The CSP may deploy FPGAs of one or more vendors. Computation capacity is rented by a *client* that communicates the workload, i.e., code and data, to the CSP. Workloads of multiple clients may share the same physical resources in the cloud, according to the allocation and scheduling policies of the CSP. The current FPGA deployment model of commercial CSPs, which is also adopted in this work, allocates an entire FPGA instance from an FPGA pool in the cloud to one client for an agreed amount of time. The protection mechanism we propose, enters into force when the CSP allocates a specific FPGA to the client.

A. Trust Model & Assumptions

Trust relations among the involved parties are as follows. *FPGA vendor:* the CSP as well as the clients trust the FPGAs and the design tools offered by the FPGA vendor.

Client-Client: co-clients sharing physical resources in the cloud are mutually distrusting.

CSP-Client: the CSP does not trust clients in general. More specifically, a malicious client may launch physical attacks remotely through rogue FPGA configurations. In such attacks, a malicious client implants virus circuits in the design intended to run on the cloud FPGA to mount various remote physical attacks, e.g., DoS attacks that could lead to shutting down cloud services.

Client-CSP: clients do not trust the CSP with their sensitive data. The CSP owns the infrastructure and is motivated by reputation and financial gain, therefore, DoS and physical attacks on the cloud infrastructure by the CSP are excluded.

Assumptions. We focus in this work on cloud FPGA configurations that intend to shut down FPGAs or perform side/covert-channel attacks using rogue primitive circuits. As such, attacks that do not deploy rogue primitives are not considered [28]. We assume that FPGA vendors are willing to support IP protection on cloud FPGAs. This is consistent with the assumptions in related work [29]–[31] and is acceptable by FPGA vendors, e.g., this is evident in Intel Stratix-10 FPGAs where Intel supports remote secure key provisioning [32]. Further, we assume the CSP, the clients, and the FPGA vendors communicate with each other over secure channels, e.g., TLS, to prevent man-in-the-middle attacks. We assume the CSP offers the clients, upon request, to run their security-sensitive applications in TEEs. Finally, we assume that standard cloud security measures and protection of software against various attacks are in place.

B. Objectives & Requirements Analysis

We summarize the objectives of this work and extract the technical requirements that allow our design in § IV to achieve the objectives under the trust relations among different parties.

Objectives. While the clients aim to protect their IP designs in cloud FPGAs, CSPs are keen to protect their infrastructure, including their FPGAs, against damage caused by rogue configurations in line with recent findings [5], [6], [8], [14]–[16].

Requirements. Based on the aforementioned objectives, we derive the following requirements.

R1: TEEs on cloud FPGAs. To protect confidentiality and integrity of the client's workload on a CPU in the cloud, CSPs increasingly offer the option to run the client's workload in a TEE (see § II), both in bare-metal instances [33], [34] and in virtual machines [35]. Analogously, such protection should be offered for workloads intended to run on cloud FPGAs. For instance, a client, who intends to run a machine learning (ML) model (trained on the client's private data) on a cloud FPGA, may want to protect the ML model against attacks that aim to extract the client's private data [36]. TEEs on cloud FPGAs can be used to achieve such protection. We refer to the components that establish the TEE on an FPGA as the *trusted shell*. Ideally, the trusted shell should be i) realized with the configurable fabric to allow for future patches, while ii) incurring minimal or no changes to FPGA architectures. This implies that protecting the integrity of the configurable trusted shell against unauthorized changes is a requirement to protect the TEE on the FPGA. Therefore, establishing TEEs on cloud FPGAs requires FPGA vendor support. This is akin to hardware vendor support to establish TEEs on CPUs, e.g., Intel SGX, and is justified, since hardware and FPGA vendors are implicitly trusted by other parties.

R2: Verifiable proof of virus-free FPGA bitstreams. One typical approach to provide the CSP with adequate assurance against known FPGA attacks is to check the client's FPGA design using proprietary tools or virus scanners [18], [19] that search for known virus signatures. For example, Amazon AWS runs vendor DRCs on the client's netlist to prevent combinatorial ring oscillatorsbefore generating the final bitstream. However, we assume that the client's bitsteam is encrypted, and hence the CSP has no access to the client's netlist or bitstream in plain. Therefore, a convincing proof must be provided to the CSP that the encrypted bitstream will pose no threat to the infrastructure as well as co-tenants.

IV. TruFPGA

To achieve the objectives in § III-B, we propose the *TruFPGA protocol*, which leverages TEEs on both CPUs and FPGAs. We further design a *trusted shell* that protects client's IP bitstream on cloud FPGAs and propose to run the design check inside the TEE, whether on the client side or the CSP side. Thus, an authentic report on the status of client bitstream can be generated inside the TEE enclave and provided to the CSP. Thus, fulfilling both requirements *R1* & *R2* (§ III-B). TruFPGA protocol, depicted in Fig. 1, consists of three phases: preparation (not shown in Fig. 1), offline (2 steps) and online (5 steps) phases.

Offline phase. In step ①, the FPGA vendor provides the client with the trusted shell, which will be installed on the cloud FPGA in the online phase, as well as with the key material that enables the protection of client bitstream. In step ②, client's encrypted bitstream is sent to the CSP to be checked for rogue circuits inside a TEE. By the end of this phase, the CSP either approves the client bitstream and proceeds with the online phase or aborts if the bitstream does not pass the virus check.

We opt for the TEE at the CSP side for the following reasons: i) CSPs are assumed to continuously maintain their infrastructures and deploy suitable defenses against known attacks, including attacks on TEEs (as discussed in § III-A), and ii) powerful computation resources on the cloud enable access to a TEE equipped with the required memory resources for the virus scanner. Nevertheless, the TEE can also be on the client side as we discuss next in § VII.

Online phase. In step ③, the CSP configures the trusted shell on the FPGA. As discussed in § III-B, the main objective of the trusted shell is to establish a TEE on cloud FPGAs. In

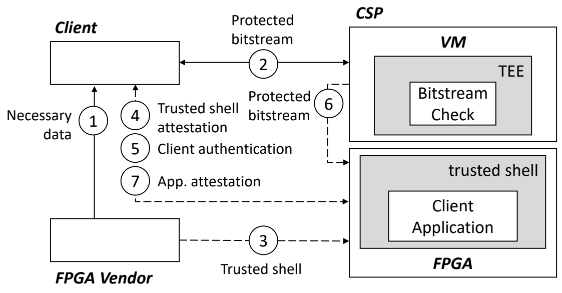


Fig. 1. `TruFPGA`: high-level overview. The straight arrows refer to the offline phase and the dashed arrows refer to the online phase of the protocol.

step ④, the client attests, i.e., verifies the integrity of, the trusted shell on the FPGA to ensure that the trusted shell has not been manipulated after configuration on the FPGA and that configuration readback is deactivated for the CSP. In step ⑤, the client authenticates itself to the FPGA to prevent unauthorized configuration of client bitstream (more details in § V). While in step ⑥, the CSP forwards the encrypted bitstream received in step ② to the intended FPGA for partial reconfiguration. Finally, in step ⑦, the client attests the intended application is configured on the intended FPGA. By the end of this phase, the client's application is ready to run on the cloud FPGA.

Next, we present our trusted shell in § IV-A and the detailed computations and communication steps of `TruFPGA` protocol in § IV-B.

A. Trusted Shell

1) Tasks: After its configuration on a cloud FPGA, the trusted shell exchanges data and receives the encrypted bitstream (for partial reconfiguration) from the CSP through a PCIe interface. The trusted shell's security-related tasks are:

Preventing read-out. The trusted shell blocks configuration memory read-out to protect the client's application. Only the trusted shell itself will have access to the FPGA configuration through the internal configuration access port.

Attestation of the trusted shell. The entire configuration memory of the FPGA, including the trusted shell, is read to compute a proof of integrity PoI_F using a nonce N_i (a random number used once in a cryptographic protocol) sent by the client and the secret key R_i. The attestation of the trusted shell is inspired by the FPGA self-attestation in [37].

Client authentication. The client must prove to the trusted shell that it knows the secret key R_{i+1} used for the encryption of the client bitstream. The client computes the proof of authenticity PoA_C over the FPGA configuration using a nonce N_{i+1} that is generated by the trusted shell and communicated to the client, and the secret key R_{i+1}. To verify PoA_C, the trusted shell also computes the proof of authenticity PoA_F and compares it to the client's PoA_C. Only upon successful authentication, secret key R_{i+1} is provided to the next task. Details on the computation of authenticity and integrity proofs are presented in § IV-B.

Bitstream verification & decryption. Only when the client authentication succeeds, i.e., when $PoA_C = PoA_F$, decryption and configuration of the client bitstream on the FPGA is permitted. The FPGA verifies the application bitstream integrity and decrypts it using the secret key R_{i+1} to obtain and configure the plain partial bitstream on the FPGA.

Some of these tasks require secret keys. We consider PUF-based secret key generation, as this approach binds the cryptographic keys to a specific FPGA instance and eliminates the need for permanent secret key storage, i.e., keys are generated on-the-fly, thus, minimizing the physical attack window. PUF-based secret key generation has been thoroughly investigated for FPGAs [38]–[41]. To prevent PUF modeling attacks [27], we use a controlled PUF (*CPUF*). The FPGA vendor enrolls the CPUF on each cloud FPGA before deployment and possesses a database of its CRPs for later use. PUF enrollment can be also performed by another trusted 3rd party.

Note that the trusted shell can be designed to accept plain partial bitstreams for clients that require no IP protection. Detailed architecture of the trusted shell is presented in § VI.

2) Design Space: Ideally, the trusted shell should be implemented in configurable logic. The advantage is two-fold; no further changes to commodity FPGAs are required and security or functional patches are feasible. This is vital, since changes to the trusted shell might be required to patch security bugs in cryptographic cores, e.g., the bitstream decryption core, as the recent work of Ender et al. [42] demonstrated an attack against the unpatchable decryption core on Xilinx 7-Series FPGAs. On the other hand, this implies protecting the integrity of the trusted shell itself prior to configuration, since a malicious party can manipulate the trusted shell to leak the secret keys. Therefore, we explore the trade-offs and propose two approaches for the trusted shell implementation, both of which fulfill requirement *R1* from § III-B and require the support of the FPGA vendor:

Fully-configurable trusted shell. In this approach, the entire trusted shell is implemented in configurable logic. To protect its integrity prior to configuration on cloud FPGAs, we rely on existing hardened authentication cores on commodity FPGAs, e.g., RSA-based authentication on Xilinx UltraScale FPGAs. The FPGA vendor enforces bitstream authentication on the FPGA, programs the public key on the FPGA before shipping it to the CSP and signs the trusted shell bitstream with the corresponding private key. This prevents the FPGA from loading an unauthorized trusted shell bitstream. We implement this approach in commodity FPGAs in § VI.

Hardening some components of the trusted shell. As an alternative approach, we propose to harden the components that perform remote attestation and client authentication tasks. These are the CPUF, the cryptographic core that computes the authenticity and integrity proofs and the finite state machine (FSM) that controls them. This further requires to harden the bus carrying the configuration data from the configuration engine to the cryptographic core to ensure that the proofs are computed on the actual configuration data. Note that the computation of the proofs can be also performed by a hardened processor/microcontroller instead of the cryptographic core and the FSM, given that the firmware implementing these instructions is protected by the FPGA vendor. The hardened components cannot be modified or altered by malicious parties, thus, they form a root of trust in the FPGA. By leveraging remote attestation, the trust is extended to the other configurable components of the trusted shell, i.e., the verification & decryption core and the configuration memory controller that controls the configuration engine through the internal interface. Note that in recent FPGA families, e.g., Intel Stratix-10 and Microsemi SmartFusion-2, hardened PUF technology is already deployed for the generation of device-specific secret keys.

B. `TruFPGA`: Protocol

This section explains the detailed computations and exchanged messages in the `TruFPGA` protocol that is presented at a higher level in Fig. 1 and detailed in Fig. 2. `TruFPGA` requires no direct communication channel between the client and the FPGA: the client uses the established secure channel with the CSP, who has full control over the FPGA, to communicate with the FPGA. Thus, the CSP has access to all messages sent to/from the FPGA.

1) Preparation Phase: **Step ⓪: PUF enrollment.** Prior to deployment in the cloud, the FPGA vendor enrolls the CPUF on each FPGA instance and collects a large number of CRPs. The CRPs are securely stored in a database at the FPGA vendor side.

2) Offline Phase: **Step ①: Acquire necessary data.** To rent a cloud FPGA, the client sends a *service request* to the CSP. The CSP then assigns an FPGA to the client according to the CSP allocation and scheduling policies. The CSP sends an obfuscated identifier of the allocated FPGA, $FPGA_{ID}$, to the client. Since the infrastructure information of the CSP is proprietary, the CSP and FPGA vendor can share a list of pseudo identifiers (PIDs) for each FPGA instance, such that a PID is never given twice. Alternatively, the actual FPGA identifier is encrypted using a secret key between the CSP and FPGA vendor, such that the client only sees an encrypted message.

The client forwards the $FPGA_{ID}$ to the FPGA vendor and gets back two messages. The first message contains the trusted shell Tsh, a nonce N_i, a challenge C_i, and a reference proof of integrity of the trusted shell PoI_V. The FPGA vendor computes the proof of integrity as follows: $PoI_V = HMAC(R_i, N_i||CD)$. Such that, HMAC is a keyed-hash message authentication code used to verify both the integrity and the authenticity of data, R_i is the secret key used in the HMAC and corresponds to the response of the CPUF to the challenge C_i: $R_i = CPUF(C_i)$, and CD is the configuration data of the targeted FPGA. The second message from the FPGA vendor to the client contains the CRP (C_{i+1}, R_{i+1}).

Step ②: Bitstream check. The client encrypts the application partial bitstream pBS_A using R_{i+1} and sends it to the CSP. An enclave on a TEE residing on the CSP side

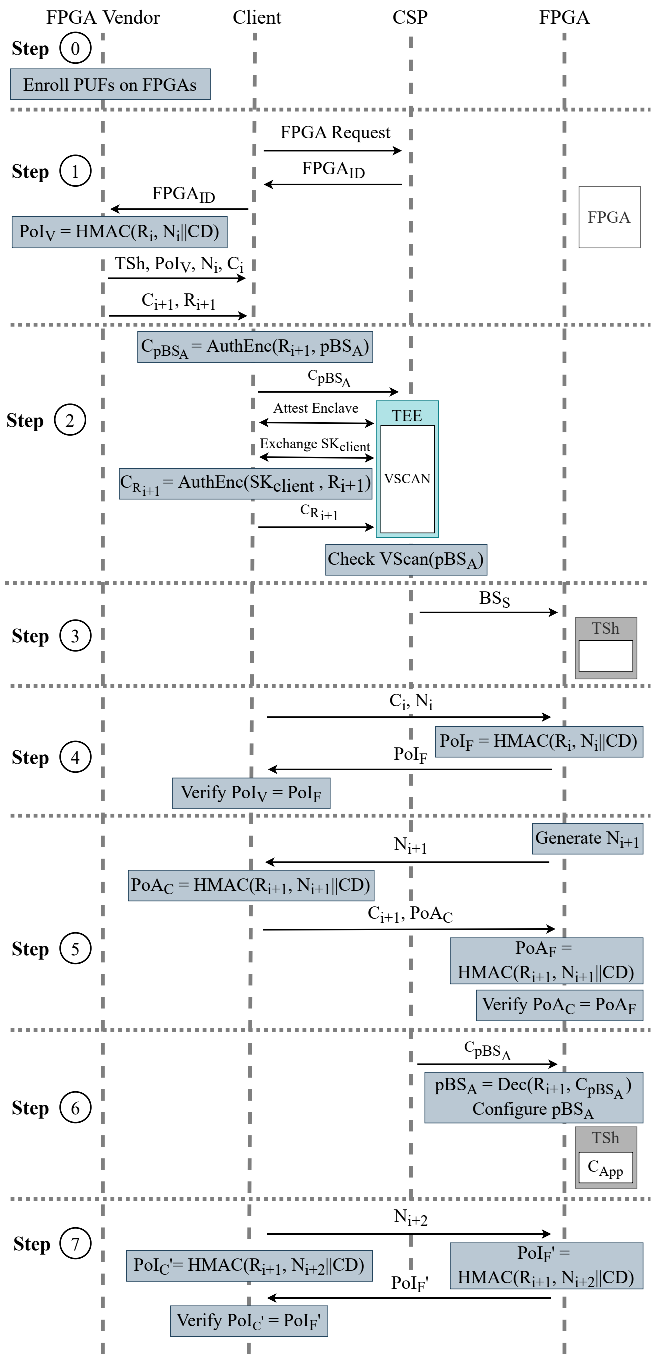


Fig. 2. `TruFPGA` Protocol. $AuthEnc()$: authenticated encryption algorithm, $Dec()$: decrypt and verify algorithm, and $VScan()$: virus scanner algorithm.

is initiated to check the application bitstream against known virus signatures. The enclave code includes a key exchange algorithm, a decryption algorithm and the virus scanner code. The client first attests the enclave binaries [43]. After enclave attestation, the client exchanges a session key SK_{client} to establish a secure link with the enclave. Through this secure channel, the secret key R_{i+1} is sent to the enclave to decrypt the application bitstream prior to the virus scan. The CSP receives only a report generated by the virus scanner about the bitstream status. The CSP has neither access to the secret key R_{i+1} nor to the plain application bitstream. The protocol proceeds if the bitstream is proven to be virus-free.

3) Online Phase: In this phase, the protocol is aborted, if any of the following steps fails.

Step ③: Trusted shell configuration. We assume the trusted shell is an open-source IP core provided by the FPGA vendor. The CSP configures the trusted shell bitstream BS_S on the intended FPGA. The trusted shell next deactivates all external configuration ports. Note that the trusted shell can be also provided by a 3rd party, therefore, it must be checked for rogue circuits before configuration on the FPGA.

Step ④: Trusted shell attestation. In this step the client verifies the installation of the trusted shell on the FPGA. Therefore, the client sends the challenge C_i and the nonce N_i, which are received from the FPGA vendor, to the FPGA. The challenge C_i is fed to the CPUF and the resulting response R_i is used to compute the integrity proof on the entire configuration memory, which contains at the moment the trusted shell only: $PoI_F = HMAC(R_i, N_i||CD)$. The CSP sends PoI_F back to the client for comparison to the reference attestation report PoI_V. In this context, remote attestation provides not only a means to verify the integrity of the trusted shell and the deactivation of external configuration ports, but also a proof of execution on the intended FPGA.

Step ⑤: Client authentication. Next, the client authenticates itself to the FPGA. For that, the trusted shell generates a fresh nonce N_{i+1} and sends it to the client. The client then computes $PoA_C = HMAC(R_{i+1}, N_{i+1}||CD)$ and sends it back to the trusted shell, together with C_{i+1}. The trusted shell computes $PoA_F = HMAC(R_{i+1}, N_{i+1}||CD)$, using $R_{i+1} = CPUF(C_{i+1})$, and verifies that $PoA_C = PoA_F$.

Step ⑥: Application bitstream configuration. The CSP sends the encrypted application bitstream to the intended FPGA. The trusted shell decrypts C_{pBS_A} into the plain bitstream pBS_A using the secret key R_{i+1}. The trusted shell then configures the application bitstream on the FPGA. As a result, the client application, denoted as C_{App}, is loaded on the FPGA and ready for use. Note that we use the same secret response R_{i+1} for client authentication and bitstream decryption to associate the bitstream to the client. Therefore, it is not possible for the CSP or other entities to load a different encrypted application bitstream after client authentication.

Step ⑦: Application attestation. As in step ③ the client attests the *entire* configuration memory of the FPGA including the application to ensure the intended application is running on the intended FPGA.

V. Security Analysis

In this section, we discuss possible attacks on `TruFPGA`. We assume that the cryptographic cores, i.e., the decryption core and the HMAC core, are cryptographically secure.

Pirated shell. To prevent the configuration of a pirated shell, whose target is to leak PUF-based keys or to compromise the computation of authenticity or integrity proofs. We discuss in § IV-A, design space, two approaches to prevent a pirated shell and to protect the integrity of the trusted shell: i) enforcing authentication on the cloud FPGA, thus only an authentic shell bitstream is configured or ii) hardening some of the components of the trusted shell.

Unauthorized configuration of client's IP. A CSP has access to all exchanged messages between the user and the allocated FPGA. After the client releases the FPGA, the CSP may use the same FPGA and instantiates it with the trusted shell. The CSP then replays prior exchanged messages with the trusted shell in order to run the client's application on the FPGA. In order to prevent unauthorized configuration of the client encrypted bitstream on the same FPGA, we add client authentication (step ⑤). In this step, the client or the party communicating with the FPGA must prove the possession of the secret response R_{i+1} used for bitstream encryption. The fresh nonce N_{i+1}, which is used to compute the proof of authenticity PoA_C, must be therefore *truly random* and is generated by the trusted shell. Note that a pseudo-random number generator (PRNG) cannot be used, because PRNG will generate the same nonce each time the trusted shell is configured on the FPGA (thus the CSP can replay recorded PoA_C). Therefore, the CSP cannot compute PoA_C since the CSP has no access to the secret R_{i+1}.

TEE security. Secure TEEs are expected to provide three properties: integrity, confidentiality, and secure remote attestation. These properties are essential to `TruFPGA` in order to ensure the integrity of the enclave code and the confidentiality of the bitstream (or the integrity of the report generated by the virus scanner in the case the TEE is assumed running on the client side, see § VII). Recent attacks on TEEs focused on extracting secret information, e.g., extracting attestation keys to spoof attestation reports [44], [45] or introducing small changes in TEE execution [46], [47]. However, influencing the computation inside a TEE in a meaningful way has not been shown yet. In response, TEE vendors are continuously patching their hardware vulnerabilities [48]. Nevertheless, TEE security is an orthogonal problem to the problem we are tackling in this paper. TEE security is an active field of research and new TEE architectures [24]–[26] are developed that try to tackle the weaknesses of current TEEs. Moreover, `TruFPGA` is TEE-agnostic and can seamlessly migrate to newer TEE technologies.

Configuration memory read-out. Major FPGA vendors allow reading back the FPGA's configuration memory after bitstream configuration and also provide the clients with the option to block this feature to prevent unauthorized read-out. However, in the cloud setting, the client has no physical access to the FPGA to enforce/ensure that this feature is blocked. Therefore, the trusted shell is designed to block configuration memory read-out after it is configured on the FPGA. Once the trusted shell is configured, the CSP cannot read out the client's application in plain form.

PUF security. As discussed in § IV-A PUF enrollment is assumed to be done by the FPGA vendor in a secure facility before shipping the FPGAs to the CSP. To prevent PUF

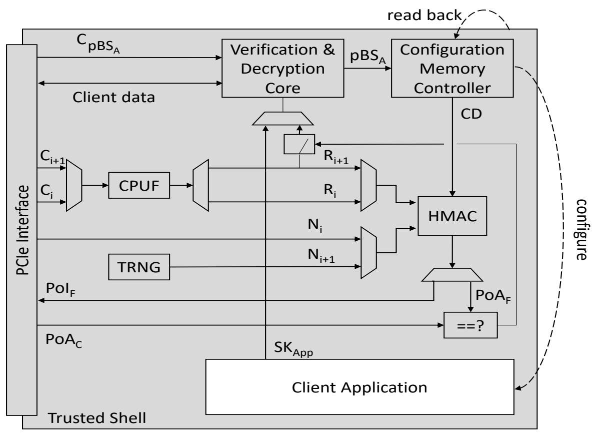


Fig. 3. Trusted shell architecture.

emulation or modeling attacks [27], we deploy a controlled PUF (CPUF) [49], [50]. As such, actual raw responses are processed internally, within the CPUF circuit, to generate the final responses. This prevents a malicious party from collecting raw CRPs for modeling attacks. Further, the FPGA vendor can set a quota of CRPs per client/day targeting the same FPGA.

VI. TruFPGA: Implementation & Evaluation

A. Trusted Shell

We prototype a configurable trusted shell, depicted in Fig. 3, on a Xilinx VCU118 evaluation board.

Components. We describe each of the components and list their required resources. The configuration memory controller is implemented using the AXI HWICAP core, which interfaces with the internal configuration access port (ICAP) primitive. The main tasks of the configuration memory controller (1161 LUTs, 1451 FFs, a BRAM, an ICAPE3) are to i) prevent configuration memory read-out ii) read the configuration memory for the computation of PoI_F & PoA_F, and iii) partially configure the application bitstream. Readback of configuration memory is disabled by setting the security bits SBITS in the Control Register to $1x$, which disables writes and reads through external configuration ports, but not to the ICAPE3 [51]. The verification & decryption core (3171 LUTs and 1004 FFs) includes an AES-128 core for bitstream decryption in the counter mode as well as a keyed hash function (AES-CMAC) for the integrity verification of the incoming client application bitstream. Note that for higher security assurance, a side-channel resilient AES core with higher security parameter (key width of 256-bit) can be used. The HMAC (1955 LUTs and 1455 FFs) for the computation of the integrity/authenticity proofs is implemented also with an AES-CMAC core with the 128-bit secret key generated by a CPUF circuit. In our prototype, we implemented a TRNG core for fresh nonces (1069 LUTs and 137 FFs) and a dummy CPUF that generates a random value for demonstration purposes only. However, we propose the use of CPUF as in [49], [50]. Commercial soft PUF IP cores are also available, e.g., Intrisic-ID Inc. (https://www.intrinsic-id.com). Note that the trusted shell must be designed to keep no records of secret keys after their usage to minimize the physical attack window. The trusted shell clears all intermediate key-related values immediately after their usage. Further, the trusted shell can be designed to configure unencrypted IP bitstreams directly for clients that do not need IP protection.

Integrity Protection. We leverage RSA authentication, which can be used independently of bitstream encryption [51], to authenticate the static trusted shell bitstream before configuration. RSA authentication is enforced by setting the OTP eFUSE Security Register (FUSE_SEC) [51], thus, only authentic bitstreams can be configured. FPGA vendor support is required to program the RSA public key and enforce the authentication prior to deployment in the cloud.

B. Virus Scanner in the TEE

In order to prove the feasibility of performing privacy-preserving checks on clients' bitstreams, we run the virus scanner on a commodity TEE with limited memory resources resembling a client machine for demonstration purposes only. We run our experiments on a computer with modest resources (Core i7-7700 CPU clocked at 3.6 GHz and 8 GB of RAM) running Ubuntu 18.04.4 LTS. We use the Graphene-SGX framework [52] to embed the virus scanner into an Intel SGX enclave [20]. We chose the open-source virus scanner FPGADefender [53], which is designed to detect short-circuits and self-oscillating circuits in bitstreams. We test our setup with a number of FPGA designs used also in [53] and implemented on a on a Zynq UltraScale+ MPSoC: parallel scrambler, stepper motor, encoder/decoder, coded decimal adder, RS_232 UART, I2C Bus, DES, AES, SHA3, PSNG, TRNG, SPI, CAN controller, Cordic, MIPS CPU, RISC-V CPU, Mandelbrot and FPGA miner. Due to the limited space, we report the runtime of the biggest tested design, SHA3 core, which is approximately 36 minutes. The runtime overhead of SHA3 compared to the runtime of unprotected version is $8.82\times$, whereas the geometric mean of the runtime overheads of the aforementioned designs is approximately $3.21\times$. This is due in our setup SGX can only access up to 128 MB of encrypted memory at a time. Therefore, for benchmarks that require more than 128 MB, the SGX driver for Linux relies on paging, i.e., least-recently-used encrypted pages are replaced when other pages are needed. Note that optimizing the virus scanner performance, as evident in [8], will significantly improve our results. To account for bigger FPGAs, e.g., with multiple super logic regions (SLR), the client can provide a bitstream for each SLR to be scanned individually. The CSP then approves the client design when bitstreams of all SLRs are scanned. To further reduce the overhead, the scan can be done once per client's bitstream. This is feasible by keeping encrypted and integrity-protected records, e.g., the scan report and a hash of the bitstream, of scanned bitstreams. The enclave then checks whether the incoming bitstream is within the list of records, by comparing its hash to existing hash values. The bitstream gets scanned, only if it is not scanned before.

VII. Discussion

Location of the virus scanner. It is also feasible to scan the bitstream in a TEE on the client side. However, it requires slight modifications to step ②. The enclave code includes the virus scanner, an encryption algorithm and an algorithm to compute $PoI_{Enclave}$ of both, the resulting scan report and the encrypted bitstream. This is required to prove for the CSP that i) the scan report has not been compromised by the client after its generation, and ii) the scanned bitstream is the one encrypted afterward. Since the TEE is on the client side, the client provides as an input to the enclave the plain application bitstream and the secret key for encryption. After initiating the enclave, the CSP attests the enclave binaries to ensure their integrity. Then, the CSP exchanges a session key with the enclave to establish a secure link with the enclave. Through this secure link, the CSP sends a secret key for the computation of $PoI_{Enclave}$. At the end of step ②, the CSP receives the encrypted application bitstream and $PoI_{Enclave}$.

CSP side vs. client side: trade-offs. In terms of *performance*, it is more efficient to run the virus scan on the CSP side than on the client side, assuming the client has access to a TEE in the first place, due to the limited computing resources of the client. In terms of *security*, compromising the TEE on the client side could result in forging the virus scan report to label a bitstream with rogue primitives as a benign bitstream. This could pose serious threats on the CSP infrastructure or functional failure of the FPGA shell. Whereas for a compromised TEE on the CSP side, the bitstream confidentiality is no longer guaranteed. This could lead to problems for companies in which IP theft has a financial impact. In principle, the CSP and the client could negotiate where the scan should happen. In practice, however, the solution where the TEE resides on the CSP side is most likely to be adopted.

Trusted shell: CSP propriety infrastructure information. We assume the trusted shell to be open-sourced to assure clients the integrity of the deployed cryptographic primitives that will process their bitstreams on the cloud FPGA and the protected access to the configuration engine. Thus, the client does not need to attest the integrity of the trusted shell interfaces (PCIe core, DRAM controller, etc.), which might be considered as proprietary information for the CSP. To achieve the goal of this work while protecting the CSP proprietary design, the trusted shell can be split into an open-sourced part (the attestable part) by the FPGA vendor and a customized part whose configuration is done through the attestable part. The customized part, which contains the proprietary FPGA interfaces, is not revealed to cloud clients. However, the customized part can be scanned for viruses by a trusted party, similar to client bitstreams in step ②, and a hash of the customized part is published instead. This is to assure the clients that the shell does not contain rogue circuits spying on their logic. FPGA configuration data of the customized part, are replaced with their hash value during the computation of the reference attestation report in step ① and the proofs computed in steps ④ and ⑤.

VIII. Related Work

To the best of our knowledge, there are few solutions that tackle IP protection in cloud FPGAs, these are thoroughly discussed in [54]. We briefly discuss the most relevant work [29]–[31]. The schemes in [29], [31] rely on an initial bitstream configured on a cloud FPGA to enable the protection and the configuration of clients' IP bitstreams. The initial bitstream contains a cryptographic core for decryption of IP bitstreams using secret session keys, which are obtained through a key exchange protocol that deploys public key cryptography, e.g., RSA or Diffie-Hellman. In [29], RSA private key is embedded in the RSA core in the initial bitstream, thus, the confidentiality of the initial bitstream must be protected and this is achieved, e.g., by leveraging the hardened AES core in Xilinx FPGAs. For that, the authors propose the FPGA vendor to program the AES secret key before deploying the FPGA in the cloud. In [31], the authors assume that the FPGA vendor configures the FPGA with the initial bitstream prior to deployment in the cloud and that the FPGA is constantly powered, even during shipping to the CSP, to maintain its configuration. In [30] the authors leverage the hardened SRAM-PUF, elliptic core cryptography and AES cores on SmartFusion-2 FPGAs of Microsemi. However, partial reconfiguration of the FPGA fabric is not available in these FPGAs, therefore the client should design and implement the interfaces to the host, which is inconvenient for the CSP.

Unlike `TruFPGA`, these solutions attempt to solve one part of the problem that is IP protection for cloud clients under the assumption of untrusted CSP. while the CSP has no assurance that the encrypted bitstreams do not include rogue circuits. Moreover, they do not prevent unauthorized configuration of client's IP by the CSP, since the CSP can replay client's messages and freely load the encrypted bitstream after the client releases the FPGA.

IX. Conclusion

We presented, discussed, and demonstrated `TruFPGA` the first trusted configuration scheme for cloud FPGAs that i) protects the intellectual property of clients through the support of encrypted bitstreams and ii) enables the cloud service provider to check the client's application bitstream for rogue circuits that could damage or disable the FPGA or attack the integrity of the cloud infrastructure. With solving both security challenges in one feasible protocol, `TruFPGA` provides the means for practical FPGA TEEs, which not only allows more clients to use FPGA cloud services but also the processing of sensitive data on cloud FPGAs.

Acknowledgment. This work is co-funded by the DFG – SFB 1119 CROSSING/236615297, the Intel Collaborative Research Institute (ICRI-CARS), the BMBF and HMWK within ATHENE, the CyberSecurity Research Flanders with reference number VR20192203, and the UK National Cyber Security Centre through the project *rFAS* (grant agreement 4212204/RFA 15971). We also thank the Xilinx University Program for providing tools and boards.

References

[1] G. V. Research, "Field Programmable Gate Array market size, share & trends analysis report," April 2020. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/fpga-market

[2] Amazon AWS, "Amazon EC2 F1," https://aws.amazon.com/ec2/instance-types/f1/.

[3] Microsoft Research, "Project Catapult," https://www.microsoft.com/en-us/research/project/project-catapult/.

[4] A. Cloud, "FPGA-based Compute-Optimized Instance Families," https://www.alibabacloud.com/help/doc-detail/108504.htm, 2019.

[5] D. R. Gnad, F. Oboril, and M. B. Tahoori, "Voltage Drop-Based Fault Attacks on FPGAs Using Valid Bitstreams," in *IEEE FPL*, 2017.

[6] K. Matas, T. La, N. Grunchevski, K. Pham, and D. Koch, "Invited tutorial: Fpga hardware security for datacenters and beyond," in *ACM/SIGDA FPGA*, 2020.

[7] C. Beckhoff, D. Koch, and J. Torresen, "Short-circuits on FPGAs caused by partial runtime reconfiguration," in *IEEE FPL*, 2010.

[8] T. La, K. Pham, J. Powell, and D. Koch, "Denial-of-service on fpga-based cloud infrastructures - attack and defense," *IACR TCHES*, 2021.

[9] G. Provelengios, C. Ramesh, S. B. Patil, K. Eguro, R. Tessier, and D. Holcomb, "Characterization of Long Wire Data Leakage in Deep Submicron FPGAs," in *ACM/SIGDA FPGA*, 2019.

[10] I. Giechaskiel, K. Eguro, and K. B. Rasmussen, "Leakier Wires: Exploiting FPGA Long Wires for Covert-and Side-Channel Attacks," *ACM TRETS*, 2019.

[11] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "An Inside Job: Remote Power Analysis Attacks on FPGAs," in *DATE*, 2018.

[12] M. Zhao and G. E. Suh, "FPGA-based Remote Power Side-Channel Attacks," in *IEEE S&P*, 2018.

[13] K. Matas, T. La, K. Pham, and D. Koch, "Power-hammering through glitch amplification - attacks and mitigation," in *IEEE FCCM*, 2020.

[14] S. Tian and J. Szefer, "Temporal Thermal Covert Channels in Cloud FPGAs," in *ACM/SIGDA FPGA*, 2019.

[15] F. Schellenberg, D. R. Gnad, A. Moradi, and M. B. Tahoori, "Remote Inter-Chip Power Analysis Side-Channel Attacks at Board-Level," in *IEEE/ACM ICCAD*, 2018.

[16] I. Giechaskiel, K. Rasmussen, and J. Szefer, "C3apsule: Cross-fpga covert-channel attacks through power supply unit leakage," in *IEEE S&P*, 2020.

[17] G. Dessouky, A.-R. Sadeghi, and S. Zeitouni, "SoK: Secure FPGA multi-tenancy in the cloud: Challenges and opportunities," in *IEEE EuroS&P, to be published*, 2021.

[18] J. Krautter, D. R. Gnad, and M. B. Tahoori, "Mitigating Electrical-level Attacks Towards Secure Multi-Tenant FPGAs in the Cloud," *ACM TRETS*, 2019.

[19] T. La, K. Mätas, N. Grunchevski, K. Pham, and D. Koch, "FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs," *ACM TRETS*, 2020.

[20] Intel, "Intel software guard extensions," https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[21] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf, 2016.

[22] ARM, "ARM TrustZone technology," https://developer.arm.com/ip-products/security-ip/trustzone.

[23] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security*, 2016.

[24] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "Sanctuary: Arming trustzone with user-space enclaves," in *NDSS*. The Internet Society, 2019.

[25] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *ACM EUROSYS*, 2020.

[26] Bahmani et al., "CURE: A Security Architecture with CUstomizable and Resilient Enclaves," 2020.

[27] U. Rührmair, J. Sölter, F. Sehnke, G. Dror, J. Schmidhuber, and S. Devadas, "Modeling attacks on physical unclonable functions," in *ACM CCS*, 2010.

[28] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar, "Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms," *IACR TCHES*, 2020.

[29] K. Eguro and R. Venkatesan, "FPGAs for Trusted Cloud Computing," in *IEEE FPL*, 2012.

[30] B. Hong, H.-Y. Kim, M. Kim, T. Suh, L. Xu, and W. Shi, "Fasten: An fpga-based secure system for big data processing," *IEEE Design & Test*, 2017.

[31] M. E. Elrabaa, M. Al-Asli, and M. Abu-Amara, "Secure computing enclaves using fpgas," *IEEE TDSC*, 2019.

[32] *Intel Stratix 10 Device Security User Guide*, Intel Corporation, 10 2020.

[33] IBM, "Data-in-use protection on IBM cloud using Intel SGX," https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[34] Alibaba, "ECS bare metal instance," https://www.alibabacloud.com/product/ebm.

[35] Microsoft, "DCsv2-series VM now generally available from azure confidential computing," https://azure.microsoft.com/en-us/blog/dcsv2series-vm-now-generally-available-from-azure-confidential-computing/.

[36] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *IEEE S&P*, 2019.

[37] J. Vliegen, M. M. Rabbani, M. Conti, and N. Mentens, "SACHa: Self-Attestation of Configurable Hardware," in *DATE*, 2019.

[38] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "Fpga intrinsic pufs and their use for ip protection," in *CHES*. Springer, 2007.

[39] A. Wild and T. Güneysu, "Enabling sram-pufs on xilinx fpgas," in *IEEE FPL*, 2014.

[40] A. Maiti and P. Schaumont, "Improved Ring Oscillator PUF: An FPGA-Friendly Secure Primitive," *Journal of Cryptology*, 2011.

[41] Y. Hori, T. Yoshida, T. Katashita, and A. Satoh, "Quantitative and statistical performance evaluation of arbiter physical unclonable functions on fpgas," in *IEEE ReConFig*, 2010.

[42] M. Ender, A. Moradi, and C. Paar, "The unpatchable silicon: A full break of the bitstream encryption of xilinx 7-series fpgas," in *USENIX Security*, 2020.

[43] C. S. Intel, "Intel software guard extensions remote attestation end-to-end example," 2016.

[44] V. B. et al., "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018.

[45] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice," https://sgaxeattack.com/, 2020.

[46] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: exposing the perils of security-oblivious energy management," in *USENIX Security*, 2017.

[47] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0LTpwn: Attacking x86 processor integrity from software," in *USENIX Security*, 2020.

[48] Intel, "Security center," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html.

[49] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Controlled physical random functions," in *IEEE ACSAC*, 2002.

[50] C. Herder, L. Ren, M. Van Dijk, M.-D. Yu, and S. Devadas, "Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions," *IEEE TDSC*, 2016.

[51] *UltraScale Architecture Configuration*, Xilinx Inc., 07 2020.

[52] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *USENIX ATC*, 2017.

[53] T. L. Kaspar Matas, "Fpgadefender," https://github.com/KasparMatas/FPGAVirusScanner, 2019.

[54] F. Turan and I. Verbauwhede, "Trust in FPGA-accelerated cloud computing," *ACM Computing Surveys*, 2020.