Runtime Protection of Software Programs against Control- and Data-Oriented Attacks

Munir Geden

Wolfson College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2022

ب

Dedicated to my beloved wife, who had a similar journey,
and to our wonderful children who cheered us up.

Acknowledgements

Firstly, I would like to express my deepest appreciation to my supervisor Prof. Kasper Rasmussen. I am grateful for his immense knowledge and our regular meetings that kept me on track. I would also like to thank Dr. Jassim Happa for his supervision during my first research project at Oxford. Many thanks to Prof. Andrew Martin and Prof. Sakir Sezer for agreeing to examine this thesis; and to Prof. Andrew Simpson for his valuable feedback during my Transfer and Confirmation examinations along with Prof. Martin. I would also like to thank anonymous reviewers whose feedback helped me to improve this work.

I owe a big thank you to my companion, my wife, Ayse, who also had her own doctoral journey along the way. I could not have completed this without your support. Another big thank you to my wonderful son, Yahya. Also, I apologise for the times that I could not play with you because of my work. You were a very patient and understanding boy, which made things easier for your parents. Likewise, another well-deserved thank you to Ali, who was born like a light into the darkest times of our world during the pandemic. Since then, you have continued to give joy to our lives with your brother. To my mom and dad, I am grateful for all your help and efforts, thanks a lot for coming to our aid, with the in-laws, whenever we were in need.

During my time at Oxford, I had wonderful friends. Making great friendships was the best part of the CDT's first year. Therefore, many thanks to fellow CDT-16 people. I am also grateful to everyone in our research group, especially to Room-303 dwellers, Angeliki Aktypi and Youqian Zhang. And the people of first floor in RHB, Jack Sturgess and Kubilay Kucuk, thank you for all your help and the conversations we had. A special appreciation belongs to the wise old Algerian homeless friend, Salah, who kept reminding me of the things that matter the most, although I do often forget.

I would like to acknowledge that this thesis was made possible primarily with the financial support of the Turkish Ministry of Education. I'd like to also mention the CDT in Cyber Security for the funding of my travel and equipment expenses. Lastly, I would be remiss not to mention CDT Admins, David Hobbs, Maureen York, and Janet Sadler; thank you for always trying to be helpful.

Abstract

Software programs are everywhere and continue to create value for us at an incredible pace. But this comes at the cost of facing new risks as our well-being and the stability of societies become strongly dependent on their correctness. Even if the software loaded in the memory is considered legitimate or benign, this does not mean that the code will execute as expected at runtime. Software programs, particularly the ones developed in unsafe languages (e.g., C/C++), inevitably contain many memory bugs. Attackers exploiting these bugs can achieve malicious computations outside the original specification of the program by corrupting its control and data variables in the memory.

A potential solution to such runtime attacks must either ensure the integrity of those variables or check the validity of the values they hold. A complete version of the former method, which requires inspection of all memory accesses, can eliminate all the performance benefits of the language used. Alternatively, checking whether specific variables constitute a legitimate state is a non-trivial task that needs to handle state explosion and over-approximation issues. Regardless of the method preferred, most runtime protections are subject to common challenges. For example, as the scope of protection widens, such as the inclusion of data-oriented attacks (in addition to control-oriented attacks), performance costs inevitably increase as well. This is especially true for software-based methods that also suffer from weaker security guarantees. On the contrary, most hardware-based techniques promise better security and performance. But they face substantial deployment challenges without offering any solution to existing devices already out there.

In this thesis, we aim to tackle these research challenges by delivering multiple runtime protections in different settings. First, the thesis presents the design of a non-invasive hardware module that can enable attesting runtime correctness on critical embedded systems in real-time. Second, we address the performance burden of covering data-oriented attacks, by suggesting a novel technique to distinguish critical variables from those that are unlikely to be attacked. This is to develop a selective protection scheme with practical performance overheads, without having to check all data variables or corresponding memory accesses. Third, the thesis presents a software-based solution that promises hardware-level protection for critical variables. For this purpose, it leverages the CPU registers available in any architecture with extra help from cryptography. Lastly, we explore the use of runtime interactions with the operating system to identify malicious software executions.

Contents

List of Figures

List of Tables

List of Abbreviations

ASLR . . . . . address space layout randomisation

IR . . . . . intermediate representation

LPR . . . . . Loop Protection Ratio

ML . . . . . machine learning

NAD . . . . . normalised angular distance

TEE . . . . . trusted execution environment

RIM . . . . . runtime integrity model

HSM . . . . . hardware security module

TPM . . . . . trusted platform module

ROM . . . . . read only memory

WSN . . . . . wireless sensor networks

TCG . . . . . Trusted Computing Group

CA . . . . . Certificate Authority

DAA . . . . . direct anonymous attestation

IMA . . . . . integrity measurement architecture

MCU . . . . . micro controller unit

MPU . . . . . memory protection unit

PUF . . . . . physically unclonable functions

ALU . . . . . arithmetic logic unit

ROP . . . . . return-oriented programming

JOP . . . . . jump-oriented programming

DOP . . . . . data-oriented programming

CFG . . . . . control-flow graph

DFG . . . . . data-flow graph

JIT . . . . . just-in-time

PRNG . . . . . pseudorandom number generator

DEP . . . . . data execution prevention

RISC . . . . . reduced instruction set computer

CFI . . . . . control-flow integrity

DFI . . . . . data-flow integrity

TCB . . . . . trusted computing base

CPI . . . . . code-pointer integrity

CPS . . . . . code-pointer separation

BTS . . . . . branch trace store

DIFT . . . . . dynamic information flow tracking

ISA . . . . . instruction set architecture

W⊕X . . . . . write-xor-execute

OS . . . . . operating system

API . . . . . application programming interface

MAC . . . . . message authentication code

PC . . . . . path constraint/condition

SW . . . . . Software

HW . . . . . Hardware

GPR . . . . . general-purpose register

FPR . . . . . floating-pointer register

RTOS . . . . . real-time operating system

RNG . . . . . random number generator

SGX . . . . . software guard extensions

FPGA . . . . . field-programmable gate array

IoT . . . . . Internet of Things

TOCTOU . . . time-of-check-to-time-of-use

CRA . . . . . code-reuse attacks

CFA . . . . . control-flow attestation

DFA . . . . . data-flow attestation

MMU . . . . . memory management unit

ABI . . . . . Application Binary Interface

DMA . . . . . direct memory access

FSM . . . . . finite state machine

API . . . . . Application Programming Interface

TPR . . . . . true positive ratio

FPR . . . . . false positive ratio

SVM . . . . . support vector machines

kNN . . . . . k-nearest neighbours

IG . . . . . Information Gain

NAD . . . . . Normalised Angular Distance

CWIG . . . . . Classwise Information Gain

CWNAD . . . Classwise Normalised Angular Distance

"There are two methods in software design. One is to make the program so simple, there are obviously no errors. The other is to make it so complicated, there are no obvious errors."

— Tony Hoare

1 Introduction

In today's world, we rely heavily on computing devices in almost all aspects of our lives, including private and public spheres, business or state-related affairs. Our civilisation has faced a radical digital transformation over the past decades, mainly thanks to the software systems running those devices like the souls of their hardware bodies. Software programs offer endless opportunities to create value for humanity by using those computing resources in different ways. But at the same time, they pose unprecedented risks to us, as our well-being and the stability of societies have become strongly dependent on their correctness.

Software correctness consists of two properties that must hold true together. The first is the legitimacy of program code concerning whether a device runs the right software, i.e., neither a malicious one nor a corrupted version. Such assurance, of course, first requires trust in the supply chain (e.g., compiler), so the programmer's rightful intention can be correctly disseminated to the device memory as an executable. The legitimacy of the executable code can be inspected in different ways. Given that most software instances do not typically change while running, a straightforward approach would be to look at their cryptographic checksums, as many attestation schemes suggest. A mismatch with the expected value would imply that the original code is corrupted. If we lack the knowledge

of acceptable values (i.e., a whitelist), checksums can also facilitate searching for a match in the malware databases (i.e., a blacklist).

On the other hand, the second and more difficult part is the correctness of the software runtime. Runtime correctness concerns whether the code in the memory is executing in the right way as anticipated. But such assurance is a challenging task that normally requires inspection of all the values appearing on dynamic memory regions (e.g., stack). Otherwise, an attacker capable of modifying those values can put the program into a state that causes the software to execute outside of its original specifications; in other words, legitimate program code can express computations beyond the programmer's intention, a phenomenon known as *weird machine*. Unlike code regions that constitute a single steady state, inspecting dynamic regions is a challenging task for two main reasons. The first one is the quick explosion in the number of states that can be observed legitimately on those regions. This makes full coverage of the state space (or searching within it) impossible, so it becomes impractical to decide whether an observed state is valid or not. Even for an attempt that concerns only branching information (i.e., control data), the number of possible paths would likely still be more than we can inspect for most software programs. The second reason is that the program inputs provided by external agents and hosted on those memory regions prevent us from knowing what a genuine state is with full precision. This is inevitable because the entire purpose of a typical software program that interacts with the external world is to perform computations based on a set of expected inputs.

To compromise the program runtime, attackers typically exploit memory bugs that are commonly found in programs developed via unsafe languages (e.g., C/C++), and make the malicious input part of the runtime state beyond the source code's abstraction. For instance, famous buffer overflow bugs can result in corruption of program variables next to the buffer. Depending on the setting, the attacker can overwrite a control variable, namely a code pointer, to hijack the control flow and make the program jump to any instruction—we refer to these scenarios as *control-oriented attacks*. Alternatively, the attacker can alter a data object,

for instance, a condition variable that decides on the execution of a privileged branch—we refer to such cases as *data-oriented attacks*. Apart from those one-shot scenarios, the adversary can systematically craft control and (non-control) data objects in the memory to perform arbitrary code execution (i.e., Turing-complete attack). For example, filling the call stack with the (return) addresses of carefully chosen instructions can achieve such an attack as long as the code provides the necessary code fragments (i.e., attack gadgets), known as *return-oriented programming (ROP)* [1]. Control-flow protections that validate branch targets [2] or ensure the integrity of code pointers [3] can reduce the number of reachable (weird) states by the attacker. However, with a suitable vulnerability, i.e., a bug that can compromise a loop with necessary branches and instructions, the attacker can still perform arbitrary code execution by modifying only non-control data objects while staying under the radar of control-flow protections.

In this thesis, the principal motivation is *to address these most challenging runtime integrity issues, both control- and data-oriented attacks in different contexts, and to make sure that a software program executes benignly as expected at runtime.* While working towards this goal, this thesis aims to provide solutions that can be adopted in practice, regarding both performance and deployability aspects.

1.1 Identified Gaps

Different studies in the literature have investigated runtime attacks. For example, well-known attack mitigation strategies, such as control flow (CFI) [2] and data flow integrity (DFI) [4] techniques, validate program executions at runtime according to a static model (approximation) via instrumentation. In the event of a deviation from the expected model, e.g., control-flow graph (CFG), the program is usually terminated. Likewise, recent runtime attestation proposals [5–7] adopt a similar understanding of the problem. But instead of performing necessary checks in real time, they provide reports about path (control-flow) or memory (data-flow) traces to a remote party that will perform the validation task later. In contrast to those methods, the alternative way is to ensure the integrity of critical runtime

objects through other memory safety approximations such as code-pointer integrity (CPI) [3]. Taking these and relevant solutions into account, we have identified the following research gaps.

Runtime Attestation without Accumulation of Trace Information. Our interest in runtime correctness originally stems from *remote attestation* problem of embedded systems, where a potentially infected device (prover) tries to convince a remote party (verifier) that the device software is in a legitimate state. Despite many proposals for code attestation, proving runtime correctness has generally been ignored. In recent years, the literature has proposed different control-flow attestation (CFA) [5–8] methods for embedded systems to reveal control-oriented attacks. In general, these methods either record path traces or digest them into a single cumulative hash value to be sent to the verifier. The verifier is responsible for checking whether the given trace or the hash can be generated by the program's CFG or not. But the former trace-based approach can suffer from costly storage and communication overheads due to the size of execution traces. In contrast, the latter method digesting those traces into a single hash measurement is not scalable to many embedded software instances since the discovery of CFG paths by the verifier might easily fail for a program containing enough nested calls and branches, which can quickly explode the number of possible paths. Another issue is that these schemes either instrument the software with costly switches to the trusted execution environment (TEE) of the system (e.g., TrustZone [5]) or request disruptive changes in the hardware architecture [6]. Therefore, they would not fit existing or legacy systems, where modifying hardware and software might not be an option.

Lightweight Data-Flow Protection. Despite many protection schemes against control-oriented attacks, the literature lacks a practical and deployable method to address data-oriented attacks, in general. Suggested data-flow integrity (DFI) [4] techniques could not be adopted in practice mainly due to their excessive instrumentation that inspects almost all memory accesses, inevitably resulting in high performance overheads. On the contrary, hardware-based data flow isolation

(HDFI) [9] not only suffers from deployment challenges but also lacks a method to identify sensitive data in need of isolation. We consider that those drawbacks can be addressed with a targeted approach that selectively instruments only critical program variables while leaving others unprotected. Such a software-based approach that avoids redundant instrumentation would enjoy better performance and adaptability.

Strong and Practical Protection for Critical Runtime Data. Regardless of the scope, in general, runtime protection methods count on the integrity of the critical data identified or created. For these protections to fulfil their promises, the data must be well protected and accessible within the same address space, as using the kernel space or a TEE domain would trigger cascading switch costs. As a software-based solution, current techniques that hide critical data in a randomised memory location can be bypassed by integrated attacks that disclose the location information first [10]. Alternative methods such as software fault isolation (SFI) [11] suffer from high overhead due to excessive checks on every memory access. On the other hand, word- and page-level hardware-based isolations [9, 12] can address stronger adversaries without any switch costs, but they are subject to expensive deployment challenges and do not offer a solution to existing devices.

1.2 Contributions

This section presents the key contributions of each core chapter aiming to fill the identified research gaps and describes what they may mean for the wider research community.

Design of a Hardware Module for Runtime Attestation. Chapter 4 proposes a runtime attestation scheme to report code and code-reuse attacks on critical embedded systems. The value of this work lies in two deliverables: The first one is *a runtime integrity model (RIM) which describes legitimate executions via a static call-graph-like model whose approximation is enhanced with a finite array of counters that represent the depth of active calls from each function.* The second is the design

of *a conceptual non-invasive hardware security module (HSM) that can enable a practical attestation by performing most validation work on the prover side.* The module is considered to be connected to the prover's bus for monitoring program execution, and reporting its measurements when requested by the verifier. Its core logic measures whether the runtime information available on the bus is in accordance with the given RIM loaded into the module's memory. Unlike previous attestation work delivering [7, 8] or digesting [5, 6] control-flow traces, our scheme avoids the accumulation of any trace information for a practical approach. Instead, it consumes the trace information in real time, thanks to local checks designed to be performed at runtime. This design choice saves us from not only potential storage and delivery costs of execution traces but also path explosion issues, which the verifier might be suffering from during the search of CFG space in the case of a digest-based approach.

Automated Distinction of Critical Variables. Chapter 5 offers a lightweight data-flow protection method to address data attacks, without having to check all memory accesses. Unlike previous attempts [13, 14] that employ programmer annotations or type-based inferences, this work introduces *a novel automated distinction between critical and non-critical variables based on the trustworthiness of their value origins.* This work considers that variables defined by trusted agents, such as the programmer, are more worthy of protection than those already controllable by untrusted agents, such as users or the environment, which can be left uninstrumented to avoid unnecessary performance costs. In addition to values defined by trusted agents, the chapter presents a static trust propagation analysis to identify other emerging values that can be extracted from those. Different isolation primitives can benefit from this analysis to create a secure or trusted domain.

Register File as Reusable Trusted Storage for Critical Variables. Chapter 6 presents *the first scheme that systematically uses existing CPU registers to ensure the integrity of critical control and data variables.* CPU registers are normally used by the compiler for performance improvement. But this chapter demonstrates that the register file can also serve as secure storage with the help of cryptography.

This chapter has two main contributions. The first is a *security-oriented register allocation* method that assigns CPU registers to program variables that are more likely to be targeted in memory, so that those variables can be protected from corruption while in use. The second contribution is a novel use of cryptographic primitives (i.e., MAC) to *ensure the integrity of the register file images spilled to the memory* as data at rest across function calls.

Malware Analysis Framework Based on Runtime Features. Chapter 7 presents *a broad empirical comparison of system-level runtime features that can be used to identify malicious executions.* Differently from other chapters, this chapter does not propose a defence against malicious executions of legitimate but vulnerable software instances. Instead, it explores the execution properties of software applications that are malicious by design and how they interact with the host system. These runtime features can be especially meaningful to address malware instances that do not allow any signature-based or static analysis methods, i.e., polymorphic and metamorphic malware. This chapter assesses the value of different feature models that can be extracted from API call traces, for machine-learning classifiers. Also, it adapts a new feature selection technique, called normalised angular distance (NAD), to a multiclass problem, in order to prioritise a feature's distinctiveness over its popularity.

1.3 Structure

The rest of the thesis is structured as follows: Chapter 2 provides the background necessary to follow the rest of the thesis, such as possible attack scenarios targeting program memory and a typical attestation protocol. Chapter 3 presents a review of relevant attestation, memory protection, and attack mitigation techniques. Following the related work, the thesis presents four main chapters on the topic. Chapter 4 delivers a runtime attestation scheme that relies on a conceptual hardware security module. Chapter 5 introduces a new method to identify critical variables without programmer annotation, and suggests a targeted lightweight software-based

approach against data-oriented attacks. Chapter 6 presents a software-based method that uses the register file as secure storage for protection of critical program data. Before concluding, Chapter 7 presents a malware analysis framework that combines the use of system-level runtime interactions with machine learning methods to identify the executions of software program that are malicious by design. Lastly, Chapter 8 summarises our contributions, discusses some of the lessons we have learned, and considers how our work may project onto the wider community.

1.4 List of Publications

The content of Chapters 5, 7, and a different attestation scheme but using a similar setting given in Chapter 4 have been presented and published at peer-reviewed conferences. The actual design presented in Chapter 4 is accepted for publication in a peer-reviewed journal, while the content of Chapter 6 is currently under review:

- Geden, M., & Happa, J. (2018, October). *Classification of Malware Families Based on Runtime Behaviour.* In the 10th International Symposium on Cyberspace Safety and Security (pp. 33-48). Springer, Cham.(Chapter 7)

- Geden, M., & Rasmussen, K. (2019, August). *Hardware-assisted Remote Runtime Attestation for Critical Embedded Systems.* In 2019 17th International Conference on Privacy, Security and Trust (PST). (Chapter 4)

- Geden, M., & Rasmussen, K. (2020, December). *TRUVIN: Lightweight Detection of Data-Oriented Attacks Through Trusted Value Integrity.* In 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). (Chapter 5)

- Geden, M., & Rasmussen, K. (Accepted for publication). *Hardware-assisted Remote Attestation Design For Critical Embedded Systems.* IET Information Security. (Chapter 4)

- Geden, M., & Rasmussen, K. (Under review). *RegGuard: Leveraging CPU Registers For Mitigation of Control- and Data-Oriented Attacks.* (Chapter 6)

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards."

— Gene Spafford

2 Background

This chapter provides the background knowledge required to follow the rest of the thesis. The chapter begins by giving an overview of a typical remote attestation protocol in Section 2.1, which is necessary for Chapter 4 to deliver our attestation scheme. Next, Section 2.2 describes possible memory corruption scenarios that can compromise the software runtime in different ways. The attack classes covered in this section are relevant throughout the thesis. Lastly, Section 2.3 explains the basics of register allocations that are fundamental to the proposal presented in Chapter 6.

2.1 Remote Attestation

Remote attestation is a challenge-response protocol that typically consists of two parties: *verifier* and *prover*. It allows *verifier*, as a trusted party, to challenge a potentially infected device, *prover*, to provide evidence that the device is in a good state, so the verifier can reason about the prover's condition. A good state can mean having the right code in the memory, and also executing that code in the right way, depending on the scope of attestation. Therefore, the literature makes a distinction between *static* and *runtime (dynamic)* attestation. The former concerns with only static memory regions such as code segments, whereas the latter group reports on dynamic memory regions that keep changing at runtime

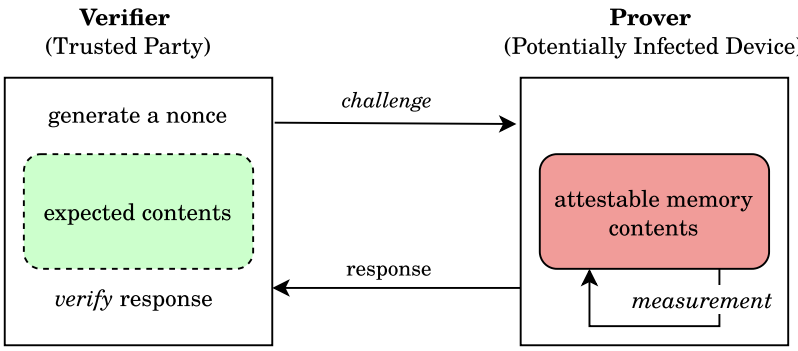


Figure 2.1: An overview of a remote attestation scheme.

(e.g., stack). Furthermore, remote attestation paradigms can vary from a pure *software-based* approach to a *hardware-based* approach that relies on tamper-resistant components such as trusted platform module (TPM). In between, *hybrid* techniques take advantage of more common hardware features, such as read only memory (ROM) and memory protection unit (MPU). Chapter 3 presents a detailed review of the schemes from each group.

Remote attestation

A challenge-response protocol between two parties, a remote trusted party, called *verifier*, requires a potentially infected device, called *prover*, to provide integrity evidence that the device is in a healthy state.

2.1.1 Protocol Phases

A typical remote attestation protocol consists of three main phases as depicted in Figure 2.1:

Challenge. The verifier first generates a random *nonce* value and sends it to the prover as a trigger for the attestation process. The nonce is necessary to guarantee the freshness and unpredictability of the attestation response. Some runtime attestation schemes [6, 15] also suggest that the verifier provides the program input to the prover at this stage.

Measurement. Upon receiving the nonce, the prover prepares a measurement to report on its state. The measurement is typically a cryptographic checksum of

attestable memory regions. Depending on the type of attestation, the measurement process can be specific to the given nonce regardless of memory contents. For example, software-based methods traverse memory addresses in a pseudorandom order defined by the nonce. Otherwise, hardware-based methods, which consider a key on the prover, sign the measurement with the nonce and respond back to the verifier with the signature, which can be generated by a MAC or digital signature scheme. In a conventional static attestation scheme, the measurement process on the prover is normally triggered by the nonce. However, runtime attestation proposals generally require continuous measurement (monitoring) to provide evidence on past program states.

Verification. When the response is received, the verifier should first be satisfied with the freshness, authenticity, and integrity of the response. Those guarantees are easily obtained by a hardware-based or hybrid system that can store a secret key on the prover to sign the measurement and nonce. In the case of a software-based approach without a key, the verifier expects to receive the response—which should be generated from a pseudorandom memory traversal—within an expected time frame following the request. Otherwise, a delayed response would be considered as suspicious (e.g., as if the program were trying to hide or reverse malicious states in the memory). Suppose that the verifier is satisfied with the response's timing, authenticity, and integrity properties. In that case, the verifier reasons about the prover's state by checking whether the provided measurements match the expected ones.

Possible threats to a remote attestation scheme can be examined in two groups. The first group consists of attack scenarios that compromise the prover's state, mainly memory. The second is the effort to prevent the verifier from being notified about those compromised states. Regarding the former, we can expect software attacks exploiting memory bugs to be the most common attack vector, considering less common physical or microarchitectural attacks. We highlight that memory attacks have also been studied outside of the attestation context. Section 2.2 visits those attack classes and well-known mitigation techniques in the literature, which

Chapters 5 and 6 also aim to contribute. Regardless of how the prover's memory can be attacked, Section 2.1.2 reviews the scenarios targeting the protocol itself, to prevent the verifier from receiving genuine reports.

2.1.2 Protocol Attacks

Considering the options [16, 17] that might circumvent the protocol's premises, a *replay* attack occurs when the adversary responds to a new attestation request using a valid response eavesdropped previously. Such an attack requires weaknesses in the challenge (nonce function), such as inadequate randomness or small value space, to break the freshness promise.

Forgery happens if the adversary can produce a valid attestation response that does not indicate genuine measurements of the prover's state. A successful forgery attack requires either a flaw in the underlying cryptography, such as the lack of collision resistance, or a stolen key from the prover.

Precomputation can be defined as the completion of the required measurements in the prover's state prior to the attestation request. As it can form a basis for a subsequent attack with the time gained, it can be combined with replay attacks for an exhaustive search of possible nonce values.

Memory copy or *hiding* attacks exploit the free space in the prover's memory to handle malicious and original code together [18]. When an attestation request is made, the adversary on the prover can still perform its checksum measurements on the expected code contents, despite the prior execution of malicious code. In case there is not sufficient space to host both original and malicious code, a *data substitution* [19] approach can be employed that keeps only the record of changes and reverts them whenever a measurement is required. For a scenario where empty addresses are filled with noise to harden memory copy attacks, *compression* [20] methods can provide extra space to host malicious and legitimate content together. Thus, valid measurements can be provided via on-the-fly decompression.

Memory copy/hiding

An attack scenario to bypass static remote attestation protocols that lack continuous monitoring. The attacker leverages free memory to store a backup of the original code while running malicious code. This is to provide the checksum of legitimate code when requested.

Proxy attacks use a more resourceful device to hold the copy of the original memory contents. The compromised prover can forward the request to the proxy node to produce a valid measurement and impersonate the prover node.

2.2 Memory Attacks

Popular low-level languages such as C and C++ provide flexible and powerful memory management features for the development of performance-critical software. However, those features come at a cost, and software systems developed using these languages inevitably host many bugs as their complexity and lines of code (LOCs) increase. These bugs might lead to memory corruptions that result in alteration of the program code or its execution. This section explains the types of common memory bugs and relevant attack classes.

Memory bugs can be grouped into two: *spatial* and *temporal*, whose corresponding memory safety solutions aim to eliminate those bugs in the first place as the root cause of attacks. The most popular example of spatial bugs beyond dispute is *buffer overflow* bugs, which can occur in both stack and heap regions. These bugs allow data to be written on a memory block (e.g., array) more than it can host, and result in corruption of adjacent memory addresses. These addresses can contain both primitive and address variables (pointers) whose alteration can constitute a meaningful attack. Buffer overflows are typically seen as *relative-address* attacks due to the linearity and continuity of corrupted addresses. However, a pointer can also go out of bounds in non-linear ways, such as *format string* bugs or modifying a data pointer first to obtain arbitrary access capability as an *absolute-address* attack.

Buffer overflow

A software bug whose exploit violates *spatial memory safety* by allowing one to write data to a buffer more than it can host, which inevitably results in the corruption of adjacent memory addresses.

On the other hand, *use-after-free* and *double-free* bugs violate temporal memory safety, which is to ensure that every memory access refers to an object that has not been deallocated. These bugs can be as powerful as spatial safety bugs and can enable an attacker to write an arbitrary value to an arbitrary location. Use-after-free errors occur when a program continues to use a (dangling) pointer after the corresponding object has been freed. Double-free errors occur when free() is called more than once. Both spatial and temporal bugs can result in modification of memory contents in unanticipated ways. Regardless of the type of bug exploited, memory corruptions can trigger different attack scenarios [21]. An attacker able to modify memory contents might have different options to express his attack. He can modify the original code or inject new code as data. If these are not possible, alternatively, the attacker can play with the code pointers to change the way the genuine code is executed. Or he can corrupt condition variables to manipulate the control-flow without deviating from the expected control-flow graph. Figure 2.2 depicts a brief overview of these options; the following sections explain those attack classes and well-known mitigations in detail.

2.2.1 Code Attacks

Memory contents are either static or dynamic depending on whether they change or not at runtime. Program code and text sections hosting constant values and strings are typically static, whereas dynamic regions such as stack and heap contain data that keep changing at runtime.

Code-corruption attacks occur when the program code does not have integrity assurance, for instance, ROM or page-based access controls (i.e., write-xor-execute (W⊕X)). These attacks modify the original code or replace it with a malicious one.

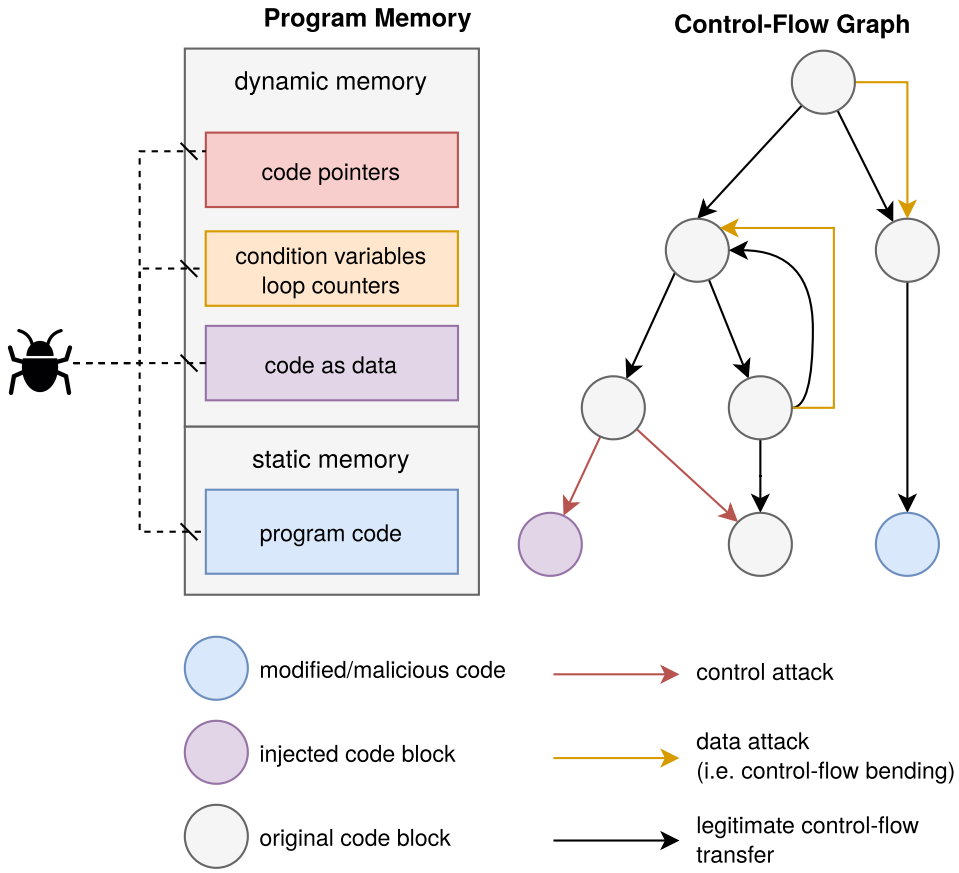


Figure 2.2: Illustration of different attack options on the program CFG.

For example, a critical jump-if-equal (je) instruction can be altered by a jump-if-not-equal (jne) opcode to execute a privileged branch. Apart from small touches, the original software can be substituted with arbitrary (malicious) code, which static attestation schemes and anti-virus/malware scanning techniques aim to address.

Code-corruption attack

A memory corruption scenario that modifies the original program code in memory to express the attack, such as changing jump-if-equal instruction with the jump-if-not-equal opcode.

Code-injection attacks inject malicious code into dynamic memory regions, i.e., data addresses, rather than modifying existing code addresses. Such an attack first requires modifying a code pointer that will point out the injected code. For instance, in a setting without canary protection [22], a simple stack buffer overflow can place a malicious payload on the stack and adjust the awaiting return address

accordingly to make the program jump to the injected payload when the current function returns. DEP or (W⊕X) mechanisms in supported architectures can prevent code-injection attacks by prohibiting any execution from data addresses (pages). Unfortunately, a typical static attestation method cannot reveal these attacks, as data addresses are not included in the checksum measurement.

Code-injection attack

A scenario in which the attacker first injects malicious code into the memory as data, and then takes over a code pointer to start executing it.

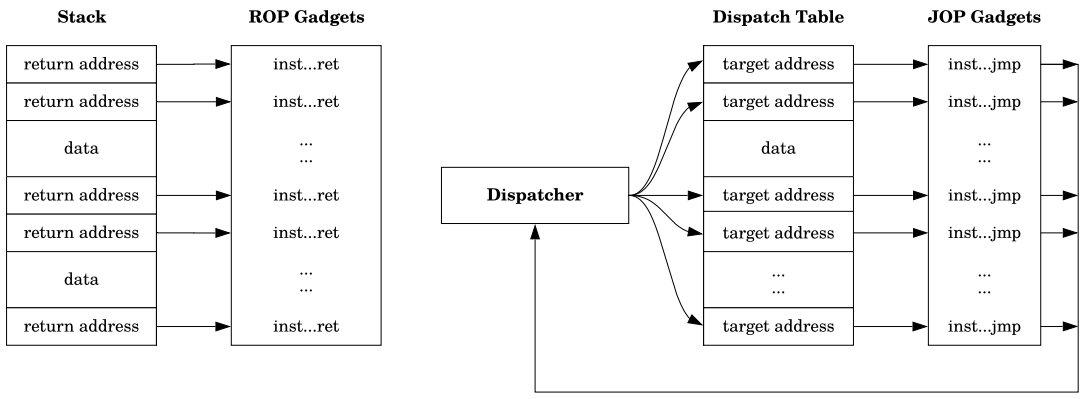
Both code-corruption and -injection attacks are mostly mitigated in today's high-end systems, thanks to the separation of code (RX) and data (RW) permissions through memory pages. However, self-modifying code cases such as just-in-time (JIT) compiled code and got/plt (global offset/procedure linkage table) sections with *lazy binding* to dynamic libraries can demand memory pages with both write and execute permissions (RWX). Those cases can be avoided using appropriate countermeasures, such as letting the JIT compiler toggle page permissions (W/X) and disabling lazy binding (`-relro`) features. On the other hand, embedded systems that run bare metal or real-time operating system (RTOS) on physical addresses generally lack these page-based permission checks, although many can still ensure code integrity through ROM features.

W⊕X

A protection mechanism that prevents a memory page from being executable and writable at the same time to counter *code-corruption* and *-injection* attacks. It requires the collaboration of both CPU and OS.

2.2.2 Control-Oriented Attacks

Control-oriented attacks, namely *code-reuse attacks*, evolved to overcome W⊕X protections. Unlike code-corruption and -injection scenarios, those attacks utilise existing code segments to express a malicious computation by carefully crafting control (code pointers) and data variables on dynamic memory. We note that code-injection attacks also need to hijack at least one code pointer as a control-flow



(a) ROP **(b)** JOP

Figure 2.3: Return- (ROP) and jump-oriented programming (JOP) attacks [26].

attack. But those differentiate from code-reuse scenarios that can systemically craft code pointers to express the attack using only existing code segments in the memory, which we describe as control-oriented attacks. Early variants of code-reuse attacks emerged as *return-to-libc* attacks [23], where the attacker calls a *libc* function with custom arguments (e.g., `system()` function used to execute shell commands). Due to the changes in calling conventions of the x64 architecture that put function arguments primarily in registers, return-to-libc attacks became less prevalent [24]. The attackers then started to use those libraries as the provider of the required code chunks (i.e., attack gadgets) until the release of address space layout randomisation (ASLR) [25] techniques that randomly arrange the base addresses of the libraries.

Control-oriented attack

A scenario that primarily alters code pointers, such as a return address (ROP) or indirect jump target (JOP), to express the attack using the genuine code segments, without being caught by W⊕X solutions.

First, Shacham et al. [1] coined the term *return-oriented programming (ROP)* to describe the systematic use of these code fragments for Turing-complete attacks. Shacham's approach is based on *ROP gadgets*, which are the code fragments ending with `ret` instructions. A ROP attack is typically carried out by crafting the stack as a chain of return addresses to these gadgets, as depicted in Figure 2.3a. Similar work by Buchanan et al. [27] has put forward the generalisability of these attacks

for RISC architectures. Although ASLR [25] has provided a probabilistic solution to *return-to-libc* attacks by making them harder to perform, ROP attacks have persisted in different ways as long as attackers could collect and locate the necessary gadgets from the program code or libraries. As a response to protections that cover only return addresses, e.g., shadow stacks, *jump-oriented programming (JOP)* [26, 28] exploits indirect jumps whose target addresses can be given from writable data addresses. First, Checkoway et al. [28] proposed these attacks. Their approach looks for a suitable *update-load-branch* instruction sequence that operates with the same effect as a return instruction on the stack. They use this type of instruction sequence as a *trampoline* to govern control transfers between JOP gadgets, which end with a jump instruction indicating which target address should be the trampoline again. Similarly, Bletsch et al. [26] have exploited indirect jumps through a designated *gadget dispatcher* and a dispatch table that defines the gadget addresses to be jumped without any requirement of the stack, as shown in Figure 2.3b. Both ROP and JOP attacks can be addressed by control-flow integrity (CFI) [2] policies that validate backward-edge (i.e., return) and forward-edge (i.e., indirect jump/call) targets according to the static control-flow graph (CFG) model. Alternatively, code-pointer integrity (CPI) [3] solutions that ensure the integrity of code pointers, rather than target validation, can also provide a practical mitigation against these attacks. On the other hand, control-flow attestation (CFA) [5, 6] schemes that count on path traces to reveal control-oriented attacks are also available in remote attestation literature. Chapter 3 discusses these and relevant protections in more detail.

Control-flow integrity (CFI)

A mitigation technique for control (code-reuse) attacks. CFI enforces branching targets to comply with the precomputed control-flow graph (CFG).

2.2.3 Data-Oriented Attacks

Data-oriented attacks, also called *non-control data* attacks, are the most challenging memory attacks today. Because they do not disturb any code or code pointers, these attacks stay under the radar of code or control-flow protections. A typical

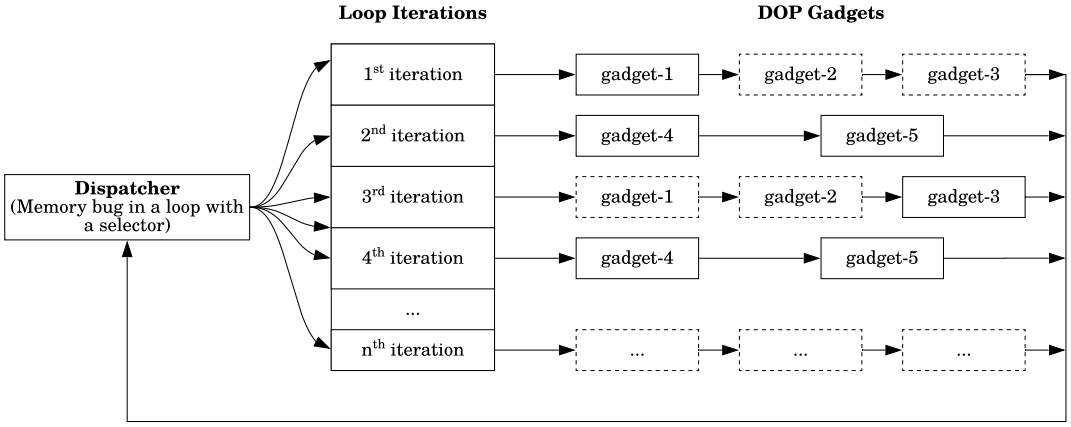


Figure 2.4: Data-oriented programming (DOP) attacks. Dotted gadgets represent instructions that do not have any impact on the aimed computation.

scenario would be the execution of a privileged branch by altering the value of a critical condition variable (e.g., admin flag). Chen et al. [29] first drew attention to these attacks by demonstrating that many real-world applications such as FTP, SSH, Telnet, and HTTP servers could be compromised under strong control-flow protection. Carlini et al. [30] split these attacks into two as *data-only* and *control-flow bending* scenarios. The former represents an attack scenario whose path (branch) trace is distinguishable from a legitimate execution, such as modifying an argument of an exec function. The latter *control-flow bending* scenarios describe attacks with path traces that comply with the control-flow graph (CFG) but cannot belong to a legitimate execution, e.g., an execution trace following an infeasible path.

Data-oriented attack

A malicious computation that is expressed via modification of (non-control) data variables, without touching any code pointers, to evade control-flow protections.

Later, Hu et al. [31] introduced *data-oriented programming (DOP)* concept and proved that these attacks can be Turing-complete in the case of a suitable vulnerability. DOP allows the attacker to execute arbitrary code by altering only (non-control) data variables, without touching any control data or a deviation from the control-flow graph (CFG). To perform a successful DOP attack, the program should contain a memory bug that can compromise a loop (the dispatcher)

with the necessary branches and instructions (gadgets), as depicted in Figure 2.4. Ispoglou et al. [32] took this a step further by automating the discovery of such a vulnerability via *block-oriented programming compiler (BOPC)*. This tool evaluates the feasibility of arbitrary code execution for a vulnerable program under strong control-flow protection. Practical mitigation of data-oriented attacks is challenging as it requires memory safety or its close approximations, such as data-flow integrity (DFI) [4]. However, these solutions generally introduce high performance overheads due to the inspection of almost every memory access. Alternative software-based solutions (e.g., WIT [33]) generally loosen the approximation precision, to reduce performance overheads. On the other hand, runtime attestation schemes that aim to reveal data-oriented attacks should provide information on the memory traces to the verifier for data-flow attestation (DFA) [7].

Data-flow integrity (DFI)

A security mechanism to mitigate control and (non-control) data attacks together. Similar to CFI, it enforces each memory write on an object to comply with the precomputed *reaching definitions* analysis, namely data-flow graph (DFG).

Information leakage scenarios that reveal unauthorised memory contents are also described by some studies as a (non-control) data attack. However, this thesis does not consider them as a data-oriented attack unless they first corrupt a data pointer. This is to distinguish them from pure confidentiality issues caused by a logical flaw in the program. We remind the reader that information leakage bugs can form the basis for integrated attacks [34] that circumvent other defence mechanisms such as ASLR [25] or CPI [3].

2.3 Register Allocations

This section explains how compilers use CPU registers to host program variables to improve runtime performance, as this is relevant to the scheme proposed in Chapter 6. Given that accessing the CPU registers is much faster than accessing the memory, the compiler usually prefers to map all program variables to the registers. However, there is no practical constraint on the number of variables that

can be defined in a program, despite the limited number of registers (i.e., usually no more than 32 general-purpose (GPR) and 32 floating-point (FPR) registers in modern architectures). Therefore, a register allocation scheme must decide how to share out registers to program variables. Thankfully, not all variables are concurrently live throughout program execution (i.e., code scope describing a variable definition to its final use). Thus, we can use registers more efficiently by assigning the same registers to different variables (i.e., live ranges) at different times. If the number of live variables is greater than the available registers at any point, this is called high *register pressure*. The compiler handles those situations by *spilling* some register values [35] to the memory or *splitting* their live ranges to create shorter scopes that could better fit both registers and memory [36, 37]. Allocation schemes usually decide which variable to be spilled using *spill costs* that estimate the candidate's number of uses and definitions at runtime, which is weighted proportionally to its loop nesting depth [38]. This aims to maximise the performance gain that the registers can provide.

Spill costs

A compile-time estimate of the performance burden of leaving a register-candidate variable in the memory.

2.3.1 Allocation Level

Register allocations can happen at a basic block, function, or program level. If the basic block is chosen as the optimisation boundary, such an allocation scheme is called *local* register allocation. Since local allocations [39] have to save and restore registers at basic block sites, they are not considered as optimal as *global* allocations happening over the whole function [40]. On the other hand, interprocedural (program-wide) allocations can only be meaningful for small programs with many short procedures [41]. Therefore, global register allocations are generally used in practice. Global allocators enable reusing the same register file repeatedly for each function call. Depending on the calling convention in place, if a register is described as a *caller-saved* register, its state is stored at the call site by the caller function.

Otherwise, the function to be called is responsible for saving and restoring a *callee-saved* register. These operations are mostly performed using simple push-pop instructions as part of the callee's prologue (save) and epilogue (restore) code.

Global register allocation

The most common register allocation strategy that shares out registers at the function level. Global allocation allows reusing registers across function boundaries as long as their states are preserved in the memory.

2.3.2 Allocation Techniques

Global schemes, which utilise registers at the function level, can adopt different approaches to solve the allocation problem. Graph colouring [35, 42, 43] is the most popular technique. It starts by building an interference graph, where the nodes represent variables and the edges connect two simultaneously live variables. The problem is formulated such that two adjacent (interfering) nodes (variables) cannot be coloured with the same colour (register). Since the given problem is NP-complete, heuristic methods are used to approximate the solution. For a node whose degree is greater than the number of available colours (registers), meaning register pressure, the compiler can evacuate some register values to the memory using spill costs that estimate the performance loss based on usage of the variables. The compiler can also iteratively transform the graph (code) in different ways for a more optimal solution. For instance, it can split a live range of a variable, which creates additional nodes that reduce the degree of a node. Or it can *coalesce* (merge) some non-interfering nodes that represent variable-to-variable operations, the total degree of which must still be less than the number of available colours (registers).

As an alternative to graph-colouring, linear scan [44] techniques aim for faster compilation times. As the name implies, they generally maintain an active list of variables that are live at the current point in the function, the intervals of which are chronologically visited to perform register allocations. This allows linear scan techniques to handle interferences without computing a graph. Those techniques [45] can especially benefit from single static assignment (SSA) features that reduce the time spent in data-flow analysis and liveness analysis, thanks

to unique variables defined on each assignment. Since naive techniques do not backtrack, they might result in less optimal allocations. However, proposals such as *second-chance binpacking* [46] address this by utilising lifetime holes (e.g., a sub-range where the value is not needed) of register values. This allows a spilled value to be placed back on a register again (splitting).

"There is no wealth like knowledge, no poverty like ignorance."

— Ali ibn Abi Talib

3 Related Work

This chapter first presents a detailed survey of remote attestation literature using a taxonomy that groups attestation studies based on the type of trust anchor, attestation scope, and interaction patterns of the parties. Then, it provides a comprehensive review of memory protections and mitigation options proposed against control and data attacks, with inline discussions of how our proposals differ from those. The chapter concludes with a brief review of malware studies using static and dynamic techniques that are relevant to our framework presented in Chapter 7.

3.1 Attestation Survey

Remote attestation schemes can be categorised in different ways. The first way is to look at the architectural features used. *Software-based* techniques do not require any special hardware features, whereas *hardware-based* methods typically leverage a tamper-resistant module as a root of trust, such as TPM, PUF, in order to address more powerful adversaries. Many schemes that employ software and hardware co-design for a more practical approach are considered as *hybrid* methods. These methods do not require tamper-resistant hardware, although they still rely on commonly available hardware primitives such as ROM. We also make a distinction between *static* and *dynamic* attestation based on the memory

scope attested. The former group typically provides checksum measurements of static memory regions, i.e., code segment, and attempts to prove that the device is loaded with the right software program, not a malicious one. Rather, dynamic attestation methods, namely *runtime attestation*, aim to convince the verifier that the software is executing in the right way, by measuring or digesting the states observed at runtime in dynamic memory regions such as stack. Lastly, remote attestation studies can be grouped based on how the verifier and the prover node(s) interact with each other, such as *many-to-one* and *one-to-many* as formulated by Steiner et al. [17]. At the end of this section, Table 3.1 summarises the attestation schemes that are reviewed based on these aspects.

3.1.1 Software-based Techniques

Spinellis et al. [47] have proposed the first software-based attestation that verifies the integrity of program (code) memory on low-end systems, without using attestation terminology. In this work, the software stakeholder, namely the verifier, asks the client device, the prover, to calculate checksums of two overlapping memory regions that cover the entire program memory together. Because the verifier sends the last address of the first region and the beginning of the second region at the time of the request, the prover would not be able to calculate a valid checksum in advance. Furthermore, the prover reports the changes in the processor's state (e.g., clock cycles), so the verifier can reveal any efforts to hide the malware (e.g., memory copy attacks).

Two similar schemes, SWATT [48] and Pioneer [19] attest program memory via time-sensitive checksum measurements. These checksums are calculated via cell-based pseudorandom (non-sequential) traversal of the prover's memory. Because a verifier-given nonce initiates the traversal, checksums cannot be calculated in advance by the attacker. Therefore, memory copy attacks that relocate malicious code to return a valid checksum should cause a noticeable delay in the response.

To prevent memory copy attacks, AbuHmed et al. [49] suggest filling empty memory addresses with incompressible noise generated based on a seed value

given by the verifier or the environment. The scheme proposes two block-based traversal mechanisms using fixed [50] and dynamic block sizes for the checksum of the memory contents, including noise. The fixed block size is determined by the verifier, while dynamic sizes are defined by the seed value provided. The authors suggest further adjustments depending on whether or not the network has synced time information. This work is vulnerable to scenarios where the original code is compressed to create usable space by the attacker.

In general, software-based attestation methods interpret any noticeable delay in the response as efforts to hide malicious code, since they rely on checksum measurements that cannot be performed in advance (i.e., prior to the verifier's request). Although relying on timing can eliminate the need for authentication of the response (i.e., attestation keys) with many practical benefits, software-based schemes make strong assumptions about the time-optimality of checksum calculations and exclude scenarios such as CPU overclocking [20] or network-related delays.

3.1.2 Hardware-based Techniques

Despite their benefits, software-based solutions are very restrictive in terms of adversary capabilities. Therefore, hardware-based methods, such as trusted platform module (TPM), physically unclonable functions (PUF), use tamper-resistant hardware modules to address stronger adversaries.

TPM is a coprocessor that serves as a strong root of trust on the prover device. In addition to the measurement functionality, TPM provides secure storage to host the necessary keys for attestation. TPM can help to verify the integrity of the software stack loaded on the device using extended measurements (i.e., PCRs) performed during the booting process, which transfers the trust from one software component to the next by creating a chain of trust. Sailer et al. [51] has used TPM for the attestation of complex software stacks via integrity measurement architecture (IMA) on Linux systems. This feature makes remote attestation possible for the whole software stack, from the BIOS to the application level. Trusted Computing Group (TCG) [52] has standardised privacy-enhanced remote attestation for TPMs

in v1.1 [53] using certificate authority (privacy CA). However, this later evolved to direct anonymous attestation (DAA) [54] in version 1.2 [54] as CAs need to be available and involved in every transaction. In contrast, DAA initially requires a one-time registration phase, where the TPM obtains a key from a DAA issuer to sign the following attestation keys (AIK) generated to preserve the privacy of the prover.

On the other hand, extensions to TPMs such as Intel TXT [55], Flicker [56] promise protected executions in addition to the measured launch of the software stack. Those extensions provide a dynamic root of trust for measurements, thanks to the isolation of security-sensitive processes at runtime. More recent on-chip trusted execution environment (TEE) solutions, such as Intel SGX [57] (enclaves) and ARM TrustZone [58] (secure world), can partition a software execution into two and can guarantee the correct execution of an attestation protocol within the trusted part. For instance, Intel SGX can protect both the confidentiality and integrity of code and data within special memory regions, called enclaves, against even privileged attackers with a compromised operating system or hypervisor.

Another hardware component used in attestation schemes is physically unclonable functions (PUF). For example, PUFatt [59] binds the manufacturing characteristics of the prover device to the attestation process to avoid impersonation attacks. This work assumes that the verifier can emulate PUFs with additional error correction proposed for robustness. In PUFatt, the verifier makes a request by sending a PUF and an attestation challenge. Then, the prover responds with a measurement generated from both challenges and the memory content, used as input to device-specific PUF—as a proof of the response's authenticity and software attestation algorithm.

3.1.3 Hybrid Solutions

Asking for a separate tamper-resistant module might be impractical and costly in some settings such as IoT and wireless sensor networks. Hence, *hybrid* solutions aim to take advantage of both hardware- and software-based approaches by utilising more common hardware features such as ROM, MPU, or FPGA.

Different lightweight architectures [60–62] have been proposed for embedded systems to ensure the trusted execution of attestation protocols. A popular scheme, SMART [60] suggests a solution for microcontroller units (MCU). SMART relies on read only memory (ROM) to store the key and the attestation code with additional access controls on the data bus through a simple memory protection unit (MPU). ROM ensures the integrity of the attestation code and keys. Given MPU monitors the program counter and allows only the attestation code to access the key. The authors have performed both a static and a dynamic analysis of the attestation code, considered part of TCB, to ensure that the code does not contain any memory bugs or leak the key accidentally. TrustLite [61] offers a security architecture for isolating critical tasks (e.g., attestation) in resource-constrained embedded systems. TrustLite guarantees both the confidentiality and integrity of the code and the data located in structures called trustlets. Thanks to the Execution-Aware Memory Protection Unit (EA-MPU) that can be programmed in software, TrustLite provides flexible memory allocations and allows updating the attestation code and the keys. Native TrustLite does not support remote attestation and allows only for local attestation. However, a similar architecture TyTAN [62] assigns a platform key, access to which is also controlled by EA-MPU. Thus, TyTAN can achieve both local and remote attestation using the designated root of trust for measurement (RTM) task. This task calculates a hash of the binary code, where remote verification uses a MAC with a key.

Francillon et al. [63] have formulated the requirements for a secure attestation process and formed the basis for the evaluation of many studies. Regarding the attestation key, the authors define *exclusive access* of the attestation code to the key with an assurance of *no leakage* afterwards. For the attestation code, the study describes three properties: *immutability*, *uninterruptibility* (atomicity of its execution), and *controlled invocation* (as an enforcement of a legitimate entry point). In the extended version of the paper [64], the authors presented a security analysis of SMART [60] showing vulnerabilities found in the scheme based on these properties.

A more recent work VRASED [65] presents a formally verified hybrid attestation architecture. VRASED suggests extending MCUs with an on-chip hardware module

that extracts information from micro controller unit (MCU), similar to what our system bus integration can obtain in Chapter 4. The proposed hardware module takes seven inputs such as the program counter (PC), memory addresses read/written by the CPU and the direct memory access (DMA) controller, and interrupt signals. The only output it can send to the MCU core is a reset signal. Similar to SMART, VRASED stores the attestation code and the key on ROM. In addition, VRASED formally verifies security requirements—which are mostly formulated by Francillon et al.'s work [64]—are satisfied for a sound remote attestation, with additional support for DMA. Nunes et al. have proposed further attestation schemes that are built on VRASED. For example, PURE [66] provides provable software update, memory erasure, and remote system-wide reset features. APEX [67] includes a one-bit execution flag with the attestation response in order to prove that the software in question is successfully executed without interruption and its output is unchanged. In order to address TOCTOU attack scenarios that can occur between two attestation requests, RATA [68] logs the latest modification time of the attestable (static) memory regions in a fixed memory address and includes the content of this address in the attestation measurement.

3.1.4 Runtime Attestation

In contrast to static code regions, a plain checksum of dynamic memory regions (e.g., stack) would not deliver much value to the verifier for two reasons: First, the verifier cannot reason about such a measurement without access to the same (external) program input with the same hardware and software settings. This is because each runtime moment or the corresponding state in the dynamic memory would be dependent on the program input given by external agents, such as environment or the user. This would make the verification problem undecidable from the verifier's perspective. Second, even if all external bits are excluded from the checksum calculation, the number of valid measurements that the verifier must discover would be still unmanageable, due to the combinatorial explosion in the number of states that remaining bits can constitute. The checksum returned would

thus be inconclusive. Unfortunately, there are countless attack scenarios that can live only in those dynamic regions, as previously given in Chapter 2. For instance, control-oriented attacks (e.g., ROP) in general can achieve malicious execution using the attack code.

Early runtime attestation work DynIMA [69] aims to address those attacks. This work extends the TPM-based IMA [51] with a dynamic taint tracking mechanism. It propagates the taint of untrusted inputs and alerts if tainted data reach a return address or function pointer. DynIMA relies on binary instrumentation and does not require the source code. A different approach, ReDAS [70] requires specific constraints to be satisfied in dynamic memory regions. It defines two classes of properties to judge the runtime integrity of a program: *structural integrity* and *global data integrity*. Structural integrity represents the fulfilment of the structural constraints of the binary extracted via static analysis. For instance, memory chunks of the heap should be adjacent because of malloc, or the return addresses are required to point to an instruction following the call instruction. On the other hand, global data integrity checks data invariants. These can be either variable values or relations that must hold during software execution, for example, a constant variable whose value cannot be changed or equality invariant of a path that forces two variables to carry the same value. The authors demonstrate the success of their scheme by testing it with nine real-world applications against known vulnerabilities such as stack overflow, heap overflow, and format string exploits.

Control-Flow Attestation. More recent studies have suggested control-flow attestation (CFA) to disclose runtime attacks that alter the control flow of the program. C-FLAT [5] and LO-FAT [6] are two pioneering schemes that implement CFA for embedded devices. On the provider side, both schemes digest path (control-flow) transitions in real time into a single cumulative hash measurement, which is later sent to the verifier. As shown in Figure 3.1, the cumulative hash function H takes the jumped node (entry address of CFG block) N_i and the preceding hash value H_{pred} as the input, i.e., $H_i = Hash(H_{pred}, N_i)$. This hash value represents the

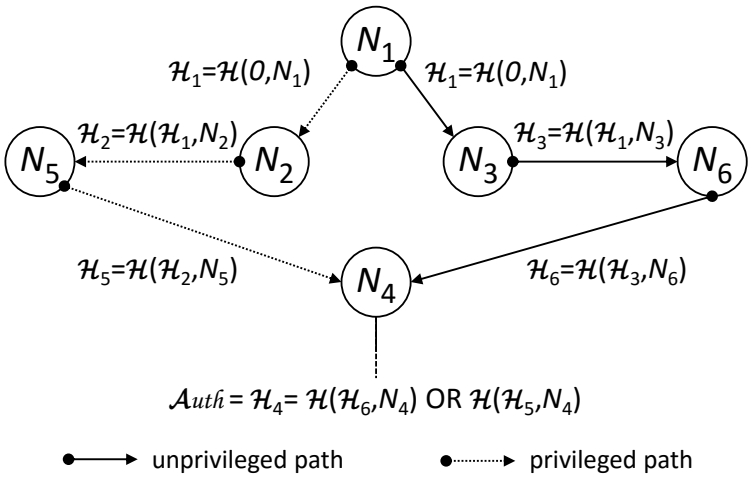


Figure 3.1: Cumulative hash calculation for control-flow attestation in C-FLAT [5].

path trace information sent to the prover. However, to make use of it, the verifier must discover all possible hash measurements, namely the corresponding path traces that can be produced from the control-flow graph (CFG). A measurement that is missing within the set of the produced hashes (traces) therefore implies an a control attack. In addition, those schemes claim that the provided hash value can disclose control-flow bending attacks, for instance, corruption of a condition variable to manipulate control-flow decisions. This attack can be described with an illegitimate path trace that is still producible from CFG. However, to make such a distinction, the verifier normally needs to have knowledge of legitimate program inputs, since static analysis looking at infeasible path constraints can only reveal a small portion of attack traces that comply with CFG. LO-FAT assumes that the verifier supplies program input to the prover. But we remind the reader that in a setting where the verifier has both the program and its input—which should typically be given by the environment in which the prover operates——actual computation task assigned to the prover could have been performed on the verifier as the trusted party. Another drawback is that both schemes suffer from path explosion in the case of a relatively complex CFG. Therefore, the verifier would not be able to complete the discovery of all CFG paths. This would also make the provided hash value inconclusive in

the eyes of the verifier. For the implementation, C-FLAT instruments the binary to represent control-flow events in the cumulative hash. C-FLAT relies on ARM's TrustZone to secure the instrumentation (hashing) process, data and keys. In contrast, LO-FAT proposes hardware extensions to the CPU for branch monitoring and hash recording. Also, to ease the path explosion issue, LO-FAT avoids the generation of a new hash for each loop iteration by separating this information.

A more recent scheme, Tiny-CFA [8] suggests secure logging of control flow events using the proof-of-execution (PoX) feature of APEX [67] scheme built on VRased architecture [65]. Normally, APEX binds the executed code to its output, stored in a configurable data-memory range. The main idea of Tiny-CFA is to instrument the code to produce append-only logs for path traces and make it a part of the output also measured. The verifier can later check this to decide whether it complies with CFG.

Data-Flow Attestation. Control-flow traces or their hash values can disclose control-oriented attacks to the verifier. But they fail to reveal data-oriented attacks generating traces that do not deviate from CFG. To address both control and data attacks together, recent schemes promise data-flow attestation (DFA). For example, LiteHAX [7] records every control-flow event in the form of encoded bitstream. In parallel, it creates a hash measurement that is computed over all the executed load-/store instructions, i.e., $H_{DF} = Hash(Load_1/Store_1, \cdots, Load_n/Store_n)$. While the control flow traces provided as bitstream enable the verifier to reconstruct the exact path executed and to attest control-flow, the hash of memory accesses allows him to verify data-flow integrity. Path traces are provided as a bitstream, so the verifier does not need to discover CFG paths this time. Instead, the verifier performs a symbolic execution of the provided bitstream and checks whether the memory accesses of the given execution produce the same hash value sent by the prover. In case of a mismatch, the verifier concludes for a data-oriented attack. Due to its online design, LiteHAX proposes communication overheads because the prover needs to send bitstreams regularly as soon as they reach a certain size.

Another scheme, OAT [71] suggests attesting control and data flow together for bare metal instances running on ARM TrustZone. OAT generates path traces only for forward edges, while using fixed-length hash values for backward edges (return addresses). This allows remote verifiers to quickly and deterministically reconstruct the control flow without suffering from path explosions. In order to avoid large memory traces, OAT locally verifies the integrity of the (non-control) data. For practical overheads, similar to our proposal in Chapter 5, OAT instruments only a small portion of program variables identified as critical, such as condition variables and the variables annotated by the programmer. As an extension to Tiny-CFA [8], DIALED [72] records program inputs using the append-only logging mechanism provided by APEX [67]. The verifier can thus check the proven control flow and the inputs logs to catch control- and data-oriented attacks together. DO-RA [15] proposes an extended CFG model as a data-oriented control-flow graph (DO-CFG) to generate hash measurements for DFA.

3.1.5 Interaction Pattern

In a network setting with multiple provers, such as WSN, remote attestation can be performed differently for efficiency and robustness. For example, Yang et al. [50] propose a distributed attestation mechanism in which the neighbour nodes of a prover device execute the verification process collaboratively, i.e., many-to-one interaction. This scheme first fills the empty memory addresses of the nodes in the network with the noise generated by PRNG and node-specific seeds, i.e., the only value needed to generate the same noise later. Later, the paper presents two attestation designs: in the first one, the prover deletes the noise seed from its memory after sharing the seed portions with the neighbours. During the attestation, if more than half of the neighbours detect some abnormal behaviours, they elect a cluster head to initiate seed recovery from neighbouring nodes. The cluster head recovers the seed via Lagrange interpolation and verifies its correctness using the seed's hash. Next, it computes a memory checksum of the node via block-based pseudorandom traversal and compares it with the prover's response. To prevent the

single point of failure in the case of a compromised cluster head, the authors suggest relying on the majority votes of the neighbours. Instead of spreading the seed among the neighbours recovered for a longer traversal, each neighbour is preloaded with some challenge (seed) and the expected response of a shorter memory traversal on the attested node. Although this design choice increases the communication overhead in the network, it provides computationally cheaper traversals.

On the other hand, USAS [73] proposes an efficient software-based attestation scheme for WSNs as an example with one-to-many interaction. USAS attest multiple provers at once by creating dynamic attestation chains. The base station, acting as the verifier, initialises the process by sending a traversal seed to a randomly picked node (I-Node). The checksum of traversal becomes the next challenge of the following nodes (F-Nodes), which constitute an attestation chain. Even though I-Node does not send its attestation result back to the base station, the base station can still detect the compromise, since only genuine I-Node responses can produce valid F-node responses.

SEDA [75] targets large-scale swarm networks where nodes can only communicate with their direct neighbours. The *operator* is responsible for the deployment and initialisation of the network devices, whereas the verifier initiates the attestation process. SEDA consists of two phases: During the first *offline* phase, the operator initialises each device by signing the certificates of code checksums ($cert\{c_i\}$) and their public keys ($cert\{pk_i\}$). When a new device wants to join the swarm network, it runs the join protocol with its neighbours to learn their code certificates, which will be necessary for later attestation stages. The online phase is triggered when the verifier makes a request to an arbitrary node with a global session identifier that is used to create a spanning tree for the session. The attestation process is performed via a recursive protocol that returns the measurement of the device and the attestation reports of its children nodes. SEDA is adapted to the SMART [60] and TrustLite [61] architectures to demonstrate how the aggregated approach reduces the cost of attesting the whole network compared to a one-to-one approach. SEDA and subsequent schemes [77, 78] using spanning trees

Table 3.1: Taxonomy of attestation schemes

Scheme	Type	Memory	Scope	Interaction
Spinellis et al. [47]	software	static	code	one-to-one
SWATT [48]	software	static	code	one-to-one
Pioneer [19]	software	static	code	one-to-one
AbuHmed et al. [49]	software	static	code	one-to-one
Yang et al. [50]	software	static	code	many-to-one
IMA [51]	hardware	static	code	one-to-one
DAA [74]	hardware	static	code	one-to-one
Flicker [56]	hardware	static	code	one-to-one
PUFatt [59]	hardware	static	code	one-to-one
SMART [60]	hybrid	static	code	one-to-one
VRased [65]	hybrid	static	code	one-to-one
ReDAS [70]	hardware	dynamic	runtime	one-to-one
DynIMA [69]	hardware	dynamic	CFA	one-to-one
C-FLAT [5]	hardware	dynamic	CFA	one-to-one
LO-FAT [6]	hardware	dynamic	CFA	one-to-one
Tiny-CFA [8]	hybrid	dynamic	CFA	one-to-one
LiteHAX [7]	hardware	dynamic	DFA	one-to-one
OAT [71]	hardware	dynamic	DFA	one-to-one
DIALED [72]	hybrid	dynamic	DFA	one-to-one
DO-RA [15]	hardware	dynamic	DFA	one-to-one
USAS [73]	software	static	code	one-to-many
SEDA [75]	hybrid	static	code	one-to-many
SALAD [76]	hybrid	static	code	one-to-many

as the communication pattern typically assume a static network whose topology does not change. For a more dynamic swarm network, SALAD [76] employs a broadcasting mechanism, where each node is responsible for propagating this request to neighbouring nodes and asking for their integrity proofs. Then, collected responses are broadcasted in the same way. The verifier can later ask any node to receive the aggregated attestation response.

3.2 Runtime Integrity

Prior to runtime attestation work, the literature had many proposals to prevent the exploitation of memory bugs or mitigate their consequences that compromise the software runtime. We can group those studies based on the line of defence where they address the problem. Suppose we ignore less common scenarios such as

physical or micro-architectural attacks. In that case, the root cause of most runtime integrity issues can be seen as the lack of memory safety features, which are missing in the languages commonly used for system programming (e.g., C/C++).

3.2.1 Memory Safety

Hence, the first line of defence should attempt to add memory safety features through runtime checks. Memory safety is typically two-fold: spatial safety and temporal safety. Adding safety features can prevent the root cause of memory exploits, regardless of how they could be used.

3.2.1.1 Spatial safety

Spatial memory safety is typically violated by a pointer that goes out of bounds. The buffer overflow vulnerability is the most popular example. Bound-checking can prevent those corruptions by ensuring that an address being accessed is within the bounds of the variable structure intended. These methods can be grouped into three: The first group is *red-zone tripwires* techniques such as Valgrind [79] and Purify [80]. Those methods use a few bits to track the validity state of each byte and place a red-zone block of invalid memory between different objects to catch continuous overflows. Since they instrument every memory instruction, their performance overheads are generally too high for use in production. Instead, they are generally used as sanitisers [81] for testing purposes, which can help detect some of the vulnerabilities. The second group of bound-checking techniques [82–84] is *object-based*. Those methods check pointer manipulations rather than dereferences. They generally keep track of object sizes on a separate data structure and ensure that each pointer arithmetic operation complies with the boundaries of the corresponding object. The third and more common bound-checking mechanism is fat pointers [85–90]. These *pointer-based* approaches implement an additional check during each pointer dereference. Using the pointer metadata provided (i.e., base, size), they check whether the address being accessed is within the range of the object. Metadata can be stored separately. For example, Softbound [88] uses a hash table

to retrieve bounds information. In contrast, Low-Fat [90] encodes boundary metadata into ineffective bits of an address word. On the other hand, hardware-based capability architectures such as CHERI [91] align the required metadata with address information. Despite the alignment of pointer (capability) data and metadata together, CHERI employs additional one-bit tags for each memory word to distinguish pointers (capabilities) from non-pointer objects. Because these tags are kept in a separate region, cache extensions are used to compensate for the performance costs.

3.2.1.2 Temporal safety

Temporal safety problems occur when deallocated object addresses are accessed later by the dangling pointers of that object. A typical example is use-after-free (UAF) vulnerabilities. After freeing a memory allocation, if the programmer does not correctly handle the pointers to that memory object, those dangling pointers can be dereferenced later to access a new object residing the same addresses. Such a case, which the attacker can exploit in different ways, has the potential to trigger data corruption, program crash, or arbitrary code execution. Temporal safety can be achieved mostly in two ways. The first group of solutions [92–94] rely on *pointer invalidation*. Those explicitly invalidate all the pointers of an object when the object is freed. For this purpose, they maintain an object-to-pointer map. The second way to achieve temporal safety is *birthmarking* [95, 96] which sets a common identifier for each object and its pointers. During memory allocations, objects are created with a random identifier value to bind both pointers and pointees. Hence, a dangling pointer holding the previous identifier cannot access the new object with a different identifier. One of the challenges of the birthmarking approach is where to store the identifiers. Disjoint structures can introduce high overheads due to additional memory access. ARM's Memory Tagging Extensions (MTE) [97], which is introduced recently, adopt a weak birthmarking approach by allocating 4-bit of tagging/colouring to every 128 bits of data. MTE leverages tagging bits to provide probabilistic spatial and temporal safety. The use of dangling pointers

should be caught because it is very likely that different bits will be assigned during the deallocation and reallocation of those addresses. Similar to CHERI, MTE extends the cache structure in order to handle the performance burden of additional memory access to tagging bits.

3.2.2 Control-Flow Protections

Most memory safety solutions incur impractical performance overheads without dedicated hardware support. Although they are used as sanitisers for testing purposes, high overheads hinder those solutions from being adopted in production. The alternative approach is selective protection of potential attack targets rather than trying to eliminate the attack vectors (bugs) that can be exploited. Normally, arbitrary code execution, i.e., Turing-complete attacks, is the most powerful scenario an adversary would desire to obtain. Thanks to the write-xor-execute (W⊕X) features, modifying the program code and injecting a custom one from data addresses is simply not an option in most systems today. But an attacker can still achieve arbitrary code execution by using the existing code segments through control-oriented attacks, namely code-reuse attacks, as we explained in Chapter 2. Although ASLR [25] shuffles existing code addresses to harden those scanarios, control-flow protection techniques are more effective. These protections can be categorised into two: *control-flow integrity (CFI)* and *code-pointer integrity (CPI)*. The former approach checks the validity of code pointer addresses and confirms whether a control-flow transfer jumps to a valid destination defined by the control-flow graph (CFG). The latter method instead focuses on the integrity of code pointers regardless of their values and make sure they are not overwritten by the attacker. Although each technique has different strengths and weaknesses, these are the most promising methods used in practice today.

3.2.2.1 Control-Flow Integrity (CFI)

CFI techniques mitigate control attacks by watching for deviations from the CFG. It has been more than a decade since Abadi et al. [2] introduced pioneering CFI

enforcement, which has been followed by many proposals [98–105]. A typical CFI is two-fold [106]. The first static *analysis* part extracts CFG [107] by approximating legitimate control transfers between program blocks (e.g., functions). The second *enforcement* part seeks compliance between the CFG edges and actual control transfers, and terminates the program in the event of an mismatch. For enforcement, CFI schemes generally use labels to bind the origin and destination sites of a control transfer.

A typical CFI scheme concerns with two types of control transfers: *backward-edge* (return) and *forward-edge* (call, jump) transfers. In case of code immutability, the attacker cannot hijack a control flow transfer whose target is hard-coded (e.g., direct call). Hence, the remaning options are modifying a return address, an indirect jump (e.g., switch statement) or an indirect call (e.g., function pointer) whose target address is given from a register or memory. Since a program function can be called from different call sites, a stateless pure graph-based return address check would be imprecise. For more precise protection, a stateful shadow stack that holds the copy of return addresses is typically used. A drawback of the shadow stack is that its size can also go unbounded like the call stack, in the case of the program with recursive functions. This is an issue especially for devices with limited resources, as Chapter 4 addressed. Regarding forward-edge transfers, CFG has to over-approximate the permissible destinations as the actual transfer can be genuinely dependent on the external input. Depending on how precisely forward-edge CFI is enforced, CFI schemes are categorised as *coarse-grained* and *fine-grained*. A coarse-grained policy describes high-level constraints, for instance, an indirect call can jump only to the beginning of a function block. In contrast, a fine-grained approach models caller-callee relations in more detail, such that an indirect call must branch to a function whose address is in the points-to set of the corresponding function pointer.

Additionally, CFI can be implemented both as a *software-based* [2, 98, 100] and *hardware-based* [99, 103, 108, 109] scheme. Software-based techniques count on instrumentation that can be performed by compiler modifications, binary rewriting, or dynamic translations. Despite the deployment advantages, software-based

methods incur higher performance costs. Although static metadata (e.g., branch labels) of forward-edge CFI can be protected by the read-only memory pages, the stateful backward-edge CFI part still relies on the integrity of shadow stack, which can be challenging to guarantee without hardware-backed solutions. On the contrary, hardware-based solutions address stronger adversaries. Despite their deployment challenges, hardware-based techniques are more favourable with better performance and stronger protection.

Software-based Approaches. Pioneering CFI proposed by Abadi et al. [2, 110] introduces an inlined enforcement system. The scheme rewrites the binary to bind two sides of a control-flow edge via identifier labels. After injecting label data into the destination sites, newly added instructions compare the label ID read from the destination address with the hard-coded one on the origin site (e.g., `cmp [DEST], ID`), and raise an error in case they mismatch. The authors also suggest using a shadow call stack to improve the precision of return integrity. MoCFI [98] targets smartphones with ARM architecture, which allows direct manipulation of the program counter and lacks a dedicated return instruction. MoCFI first extracts the application CFG provided as a patch file that is part of the load-time module, whereas CFI enforcement is handled by the runtime module. Unlike the original CFI [2], the load-time module replaces all relevant branches with a single dispatcher instruction which redirects all control tasks to the runtime module. MCFI [100] offers a modular CFI [2] using a separate compilation feature that allows the application modules to be independently instrumented and linked. Similar to CFI [2], MCFI creates labels to represent control transfer destinations where multiple targets of an indirect branch should fall into the same class. Differently, the label data is located outside the code section and handled by two separate tables stored in a runtime structure. Apart from the research studies, Microsoft's Control Flow Guard [111] enforces CFI by embedding CFG information at compile time using Visual Studio features.

Hardware-based Approaches. Davi et al. [108] present a hardware-based CFI for embedded systems. The authors suggest extending instruction set architecture (ISA) with two new instructions. The first CFIBR instruction placed at function entries manages the active label ID to the *Label State* memory serving as a shadow stack. The second post-call CFIRET instruction confirms that the returned site belongs to the active label/function. In addition, the scheme proposes to catch JOP attacks via some heuristics such as the count of consecutive jumps and pop/push instructions (i.e., deciding for an attack in case of five indirect jumps in a row). Despite the lack of a prototype, the authors share their implementation plans for Intel Siskiyou Peak processor organised as a 32-bit Harvard architecture. A more finished version of this work, HAFIX [103] enforces a fully precise backward-edge CFI using Label State memory. Although HAFIX does not check indirect jumps, it guarantees that each indirect call jumps to a function entry at least as a coarse-grained policy. In addition, HAFIX handles recursion with a counter register that keeps track of the depth of the recursive function without asking for an unbounded Label State memory. It should be noted that this counter feature works only in case of a regular recursion (i.e., a function calls itself) and do not cover indirect recursion scenarios, unlike our solution in Chapter 4. HAFIX is implemented on both Intel (Siskiyou Peak) and SPARC (LEON3) processors. Another promising work, HCFI [104] modifies SPARC ISA and establishes a shadow stack for return addresses within the CPU. HCFI promises to detect not only ROP attacks but also JOP scenarios with the help of fine-grained forward-edge labels. Although a CPU-based shadow stack is a more secure option compared to a memory-based alternatives, it poses some challenges in multi-threaded environments by making context switches very costly. Apart from these academic proposals, hardware-based CFI has also been recently deployed in practice. For instance, Intel's Control-Flow Enforcement Technology (CET) [109] offers a coarse-grained forward-edge and a strong backward-edge CFI protection with shadow stack. CET does not promise much for the indirect calls, i.e., ensure that they can only jump to the beginning of function blocks. On the contrary, to protect its shadow return addresses (stack), it designates memory pages accessible

only to special memory instructions inspected by the memory management unit (MMU). Similarly, ARM's Branch Target Identification (BTI) [112] employs a coarse-grained forward-edge CFI using a two-bit labelling mechanism called landing pads. BTI avoids type confusion among different branch types. And it prevents the attacker from using an indirect call as an indirect jump, or vice versa, in order to reduce available JOP gadgets. Unlike our solutions, the main drawback of all these hardware-based control-flow protections, they pose substantial deployment challenges and do not offer any solutions to existing devices.

Probabilistic Approaches. In contrast to graph-based approaches, Zhang et al. [113] suggest a probabilistic control-flow validation for embedded systems. The authors introduce an anomalous path checking mechanism to detect attacks exploiting multi-target jump instructions (i.e., conditional and indirect jumps). The study relies on the analysis of sliding windows, where each window corresponds to a *n-jump path* vector extracted from the execution traces. The proposed work first performs training runs in a safe environment to define feasible n-path vectors using different program inputs. Then, it implements a control mechanism in the CPU pipeline to check whether the actual execution complies with extracted path data. CFIMon [114] similarly performs a probabilistic validation of indirect jumps. It uses the branch trace store (BTS) mechanism that is available as a performance counter for modern CPUs. CFIMon consists of two phases which are *offline* and *online*. During the offline phase, it first scans the binary to create sets for each control transfer type, excluding direct calls and direct jumps. For validation of return and indirect calls, the scheme uses *ret_set* and *call_set*. The former set contains the following addresses of call instructions, whereas the latter set has the entry addresses of program functions. For indirect jumps, they additionally apply training runs to create a set of possible jump targets, namely *train_set*, where an indirect jump branching to an address that is not within this set is described as suspicious. As a combination of both CFG and probabilistic models, Xu et al. [115] use a statically initialised learning model to detect stealthy attacks that might

cause invalid (code-reuse attacks) or abnormal (non-control data attacks) control flow. First, the authors extract the CFG of each function, which is represented by a matrix containing also estimated transition probabilities. The study then aggregates individual function matrices into a larger matrix to obtain the call transitions of the entire program. The aggregated matrix is used to initialise the probabilistic model of program behaviour as a hidden Markov model. The model parameters are tuned via training runs and used to catch anomalies. Shu et al. [116] similarly present an anomaly detection based approach that analyses program behaviours using large-scale execution traces instead of applying short-sized windows on them. The primary goal of the study is to detect aberrant path attacks, which do not deviate from the CFG nor use suspicious arguments. The authors define two common anomalous patterns for the detection of those attacks. The first one is *montage anomaly* which represents the composition of multiple control flow fragments that are incompatible in a single execution. The second is *frequency anomaly*, where aberrant ratio/relations are observed between/among event-occurrence frequencies. The authors first partition the program activities via call instructions (returns are not excluded to avoid duplicated correlation of call-return pairs) to create two matrices representing event co-occurrences of call routines and transition frequency between them. The proposed model consists of two phases. The first training phase is responsible for profiling legitimate program executions via behaviour clustering. The second detection phase watches for the co-occurrence of events and occurrence frequencies on the traces to detect program anomalies.

3.2.2.2 Code Pointer Integrity

Alternative to CFI schemes that validate branch targets, namely code pointer values, code-pointer integrity (CPI) techniques can protect the control flow by preventing attackers from modifying code pointers. The original CPI [3] scheme splits the program memory into two with a *regular region* and a *safe region*, where the latter region hosts sensitive pointers that can be corrupted for a control-flow attack. Sensitive pointers are described as code pointers and data pointers that can

be used to access the code pointers indirectly. The separation of sensitive pointers is mainly achieved by a static analysis that is supported by a runtime mechanism, which handles data pointers that can access both regular and safe objects. As a relaxed variant of CPI, the authors also suggest code-pointer separation (CPS) that only includes code pointers amd leave data pointers that may access code pointers unprotected. For the protection of safe regions, the original CPI randomises their locations within the same address space as a software-based approach, which is susceptible to integrated attacks that can disclose those locations. However, different isolation mechanisms such as HDFI [9] and IMIX [12] also port CPI as a hardware-backed solution to address stronger adversaries.

Cryptographic Approaches. Isolating code pointers in a protected memory region is not the only option. Similar to our proposal presented in Chapter 6, there are recent cryptographic proposals [117–119] that leverage message authentication code (MAC) primitives to ensure code pointer integrity. For these schemes to work correctly, the MAC key must be kept confidential in a register that must not be saved in the same address space. Although these schemes do not worry about the scenarios targeting shadow or safe stacks, they need to address pointer substitute (or replay) attacks. CCFI [117] is the first scheme that has used MAC for control flow protection on x64 architectures. A CBC-MAC is computed and placed along with each control object in memory. To harden replay attacks, CCFI extends each 48-bit code pointer to a 128-bit AES block with additional parameters (e.g., frame address). CCFI leverages Intel's AES-NI extensions for speed-up. CCFI uses 11 out of 16 FPRs (XMM) to store round keys so that they do not cause performance overhead due to scheduling every time. ARM's recent pointer authentication (PAC) [112] extension provides a hardware accelerated MAC (e.g., QARMA [120]) implemented as a single instruction to check the integrity of code pointers (also data pointers if needed) with practical overheads [118], i.e., around 1% overhead for code pointers (and 20% for data pointers). PAC tags are generated from effective address (32-51) bits and squeezed into the upper (ineffective) part (11-31 bits) of the word not

required by virtual address configuration, which might make them susceptible to brute-force scenarios depending on the available number of bits. PAC associates return addresses with the stack pointer to harden replay (pointer substitute) attacks. Lastly, ZipperStack [119] creates a chain of MACs for return addresses on the stack. This study protects only return addresses and does not cover other control hijack attacks that leverage indirect branches. Similar to PAC, ZipperStack stores MACs in the upper (24-bit) space of the word, providing weaker protection. Apart from their limited attack coverage, none of those cryptographic schemes leverages the security and performance features of CPU registers together as means for protecting control and data variables in use, as our proposal in Chapter 6.

3.2.3 Mitigation of Data-Oriented Attacks

Control-flow protections fall short of mitigating data-oriented attacks that manipulate program runtime without touching any code pointers, namely non-control data attacks. For example, an attacker can corrupt a condition variable to make the program jump to a privileged branch. To mitigate those attacks, many methods are proposed in the literature, such as data-flow integrity (DFI) [121], write integrity testing (WIT) [33] and more targeted approaches [14, 71]. In general, these schemes approximate memory safety better than control-flow protections with wider coverage. However, none of those have been deployed in practice for different reasons, such as high performance overhead or new hardware requirements.

3.2.3.1 Data-Flow Checking

Miguel et al. [121] introduced the pioneering data-flow integrity (DFI) scheme to address control and (non-control) data attacks together. Unlike CFI, DFI first extracts a static data-flow graph (DFG) of the program using the *reaching definitions* analysis. This is computed by combining a flow-sensitive intra-procedural analysis and a flow- and context-insensitive inter-procedural analysis. Reaching definitions of a memory read instruction (variable use) is formulated as the set of instructions that can legitimately write the target address (variable definition). Following the

DFG extraction, DFI instruments memory accesses to check the compliance between static and runtime information. The authors maintain a *runtime definitions table (RDT)* to log the identifiers of the most recent instructions that define each data address (SETDEF). When the same address is read, the recorded identifier is checked to confirm that the address has been previously written by an instruction in the set of statically computed reaching definitions (CHECKDEF). DFI incurs high runtime (104%) and memory (50%) overheads.

HDFI [9] proposes a similar but more coarse-grained hardware-based approach. Instead of checking fine-grained instruction identifiers, HDFI separates program data into two using one-bit tags that label each memory address as either sensitive (e.g., code pointers, private keys) or non-sensitive. These tags are later used to enforce data-flow policies, such as the Biba (integrity) and Bell–LaPadula (confidentiality) models. Tags are kept in a separate tag table supported by cache extensions. HDFI extends ISA with three new instructions (sdset1, ldchk1, ldchk0). sdset1 is used to set the tag when a sensitive memory object is written (e.g., code pointers, private keys) while regular store instructions unset the tag bit. Therefore, ldchk1 aligned with anticipated reads of a sensitive object can reveal corruption scenarios, as the unset tag would imply that the object is overwritten by an unauthorised instruction. On the other hand, ldchk0 can catch information leakage scenarios by forcing all regular load instructions to read only untagged data. Unlike DFI, HDFI isolates these domains from each other and excludes scenarios where the memory instruction of a sensitive object accesses another sensitive object.

HardScope [122] proposes a two-fold hardware-assisted intra-program data isolation. The first *runtime enforcement* component ensures that the static variable/lexical scope information defining the visibility of variables to different code blocks is maintained at runtime. The second *context-specific enforcement* handles dynamically changing rules for each invocation of a code-block at runtime. For the runtime enforcement part, HardScope modifies the compiler and ISA to implement memory access rules. To handle context-specific dynamic rules, Hardscope employs a data structure, called *Storage Region Stack*, in hardware-isolated protected

regions. This region consists of frames corresponding to different execution contexts (i.e., active instances of functions), where each frame can handle different access rules.

Taint Tracking. Taint-tracking inspects how potentially malicious input propagates throughout the program to check whether it flows onto any sensitive object or a potential target. Suh et al. [123] deliver a hardware-based dynamic information flow tracking (DIFT) system to prevent the use of tainted data as code pointers. DIFT keeps track of the untrusted (potentially malicious) data that flows through memory addresses and registers. DIFT requires modifications of the whole architecture in order to extend each memory word with a one-bit tag. This includes disruptive changes in all relevant components, such as the bus and cache. DIFT propagates the taint at the instruction level via prescribed rules for arithmetic, copy, and address dependencies. Despite the strong protection against both control and (non-control) data attacks, DIFT has not been deployed in practice because of mainly its architectural challenges.

Software-based dynamic taint tracking frameworks [124–126] have also been proposed in the literature. Those schemes are typically used for testing purposes because their performance overheads prevent them from being adopted in production. For instance, TaintTrace [125] offers a promising dynamic taint tracking system. It applies additional optimisations to reduce performance overheads. The system mainly consists of four components: *configuration file* containing security policies, *shadow memory* structure that stores application data taint information, *program monitor* responsible for tracking and checking the policies, and *loader* locating these components on the memory before giving the control the program monitor. Additionally, TaintTrace allows users to specify critical paths in their program where control flow should not be dependent on tainted data to address some of the non-control data attacks.

Despite being less popular, static taint analyses [127–130] aim to eliminate runtime overhead by analysing the program at compile time. Static techniques can recognise logical flaws, such that tainted data reaches a sensitive function

(i.e., taint sink) without sanitisation. However, detection of scenarios that can modify memory addresses beyond the foreseeable program semantic is a non-trivial task. Furthermore, taint-tracking techniques may ask for intervention to define taint sources, sanitising functions, and taint sinks. Differently, Chapter 5 presents a static trust propagation scheme which can be considered as the negation of taint-tracking systems. Our proposal combines both static and dynamic techniques as a more hybrid approach. The chapter first identifies untaintable variables mostly in an automated way. Then, it selectively instruments memory accesses to those to detect runtime problems not foreseeable by the program semantic.

3.2.3.2 Pointer-Based Checking

Write Integrity Testing (WIT) [33] is a software-based mitigation technique that validates critical pointer dereferences using an interprocedural version of Anderson's points-to analysis [131]. Unlike DFI, WIT instruments only unsafe memory writes as the root cause of most attacks. Thanks to the memory reads and safe objects skipped from the instrumentation, WIT provides overhead advantages compared to other methods. Because WIT validates memory writes (pointer accesses) using a compile-time analysis, it provides less precise protection than bound checking mechanisms that can update their pointer metadata (bounds) at runtime. In other words, the attacker can manipulate the pointer address to a different memory object as long as the object is within the points-to set.

On the other hand, two relevant studies PointGuard [132] and data space randomisation (DSR) [133] mask pointer addresses with random values and unmask them prior to their use. Their main goal is to make pointer corruption useless or unmanageable for an attacker who cannot know masking values. Because pointers can legitimately access different variable addresses, those studies divide variables into equivalence classes based on *points-to* sets, so they can be masked with the same value. The number of equivalence classes depends on the precision of the pointer analysis. Hence, both schemes are subject to similar precision issues like WIT. In addition to PointGuard, DSR masks non-pointer variables

such as integer variables. Although masking can significantly harden pointer-based attacks (becomes harder as the address space increases), masking non-pointer variables might not provide meaningful protection, especially considering the branch decisions made based on boolean or value range comparisons. Apart from address validation or masking techniques, data pointer integrity can mitigate most data attack scenarios. If complete pointer integrity were obtained, the attack surface would be significantly reduced without arbitrary memory write or read capabilities, where the only attacker option would be overflowing an array to an adjacent non-pointer target (e.g., condition variable). As mentioned earlier, ARM's PAC [112] extension can be leveraged to ensure the integrity of all pointer variables in return for higher performance costs [118]. Unlike those, Chapter 6 not only guarantees the integrity of pointer variables on the stack, but also protects integral variables that can be leveraged for a relative-address attack.

3.2.3.3 Targeted Approaches

In contrast to control-flow protections, proposed techniques for data-oriented attacks approximate memory safety more closely by auditing almost every memory access. However, software-based approaches either incur high performance overheads (e.g., DFI [121] with 104% overhead) or loosen the approximation accuracy with more coarse-grained checks (e.g., WIT [33] as a flow-insensitive solution) to reduce overheads. On the contrary, it is difficult to deploy hardware-backed solutions [9]. Alternatively, selective instrumentation of only critical program data can be a promising approach as a software-based solution. However, as addressed in Chapter 5, identifying critical program variables is a challenging task because deciding on the criticality of a variable normally requires a semantic understanding of the program, without programmer annotations.

Regarding the targeted approaches proposed in the literature, KENALI [13] protects the Linux kernel against specific data-oriented attacks. Apart from the control data, KENALI defends non-control data responsible for the kernel's access control mechanisms known as reference monitors. The proposed scheme consists

of two phases. First, it identifies the regions that host essential data for security checks. To achieve this, KENALI searches the kernel code for return instructions that may return specific security error codes (i.e., -EACCESS). Then, it traces back to the branch variables used as decision-making data for these returns, which are identified as distinguishing regions. KENALI not only prevents non-distinguishing data from flowing to distinguishing regions, but also provides protection within the distinguishing regions using WIT [33]. KENALI specifically targets kernel security and mitigates only privilege escalation scenarios as a subset of possible non-control data attacks. Furthermore, it does not propose a generalisable automated solution for user applications as it requires an understanding of the program. For user space programs, Datashield [14] allows the programmer to annotate custom data types (structs) as critical, in order to isolate the data held by those. In addition to the impracticality of the annotations, Datashield associates the data sensitivity, which is ideally the property of the value, with the custom data types. This type-based approach overlooks the critical data kept on generic types (e.g., integer). OAT [71], mentioned as a runtime attestation scheme in Section 3.1, suggests checking only condition variables and their backtracked dependees as part of its *critical variable integrity (CVI)* scheme. Similar to our work in Chapter 5, CVI confirms the integrity of a variable def-use chain using variable values instead of instruction identifiers as DFI [4] suggests. CVI requires the programmer to annotate other critical variables that are used by sensitive functions without being part of any control variable slice. We highlight that CVI would unnecessarily instrument condition variables that are legitimately defined by the user or environment. Differently, our proposal in Chapter 5 decides on the criticality (need for integrity assurance) of program data based on the trustworthiness of their value agents. Our approach not only operates on the underlying cause of being integrity-critical but also allows for a more fine-grained and automated selection of critical data.

3.3 Malware Analysis

Checking the correctness of program runtime based on static control- or data-flow models can provide effective mitigation against memory exploits. However, the deployment challenges of those mitigations or the inevitable holes left in the attack surface (due to over-approximation limitations issues) can eventually lead to malware installation. In such a case, malware analysis or an anti-virus program is required to identify the malicious software. Malware analysis is generally grouped into two categories, static and dynamic techniques. The former static approach typically scans the program code for specific opcode sequences or strings without running it. On the contrary, the latter dynamic methods use features or call patterns that are recorded at runtime by executing the suspect program in a controlled environment, similar to our work in Chapter 7.

3.3.1 Static Techniques

Detection of malware based on static features and machine learning (ML) techniques was proposed long time ago. Following the first shift from heuristic-based methods to ML techniques triggered [134], Abou-Assaleh et al. [135, 136] have suggested using ngram bytes extracted from binaries as ML features to represent benign and malicious instances. This study has used the k-nearest neighbours (k-NN) algorithm for the classifications. Similarly, Kolter et al. [137] have used ngram bytes to detect malware instances on Windows systems. This work selects the most informative five hundred 4-grams as binary features, which are ranked based on their information gain scores [138]. The study is reported to have achieved 0.98 true positive ratio (TPR) and 0.05 false positive ratio (FPR) using 10-fold cross-validation with the Weka classifiers. To improve these, Reddy et al. [139, 140] have proposed two alternative ngram/feature selection methodologies, which are class-wise document frequency [139] and episode discovery algorithm [140]. Another study conducted by Santos et al. [141] also uses ngrams to score the similarity of unknown instances to known classes. Additionally, the study uses a distance parameter to tune TPR and

FPR metrics during classifications. The authors have also performed experiments to determine the optimal size of ngrams and k of k-NN algorithms.

3.3.2 Dynamic Techniques

As an example of dynamic malware analysis that counts on runtime features, Salaehi et al. [142] leverage the traces of Windows API calls collected via the *WINAPIOverride32* tool. The study uses document frequencies to rank the most representative features, where each instance is given as binary feature vectors to the Weka classifiers. The authors have used Adaboost meta-classifier option with 10-fold cross-validation and have successfully detected malware with 98.4% accuracy. Another work by Uppal et al. [143] also uses API calls collected by the *APIMonitor* tool. Differently, this study extracts ngrams from API call logs. For feature selection, the authors use *odds ratio* to decide on representative features. Similar to the previous study, the authors train the classifiers with a 10-fold cross-validation resulting in 98.5% accuracy for SVM and 4-grams. However, in the light of their methodology descriptions, both studies seem to be subject to overfitting bias as the authors apply the feature selection independently before the cross-validation phase.

Instead of (binary) detection task, MEDUSA [144] classifies metamorphic engines using the frequencies of API calls made. The authors rely on statistical measures to distinguish metamorphic engines with the help of signature vectors created for each. These vectors are created on the basis of the average frequencies of selected critical API calls for the given engine. Other studies [145, 146] achieve category and family identification by extracting features from API calls in different ways and using other artefacts such as DNS requests and accessed file names. Pirscoveanu et al. [145] categorise malware samples based on types such as Trojans, Worms, Adware, and Rootkits. The authors generate features by combining API sequences, API frequencies, and other artefacts such as the level of DNS requests. The study reports that the experiments have generated 0.896 TPR for the identification of categories. Similarly, Hansen et al. [146] detects malware and distinguishes their families with a similar feature representation, including the API arguments in the model. This

study achieves 0.864 TPR for five families that have different functionalities and components, which can be suffering from category bias. Similar to Chapter 7, both studies use the runtime reports generated by *Cuckoo Sandbox* [147]. They achieve the best accuracy with Random Forest classifier.

Another Cuckoo–based work [148] proposes a dynamic analysis framework that is supposed to be resilient against evading mechanisms of polymorphic and metamorphic malware. They use WinAPI calls and files as features. To calculate the similarity between the call traces of two samples, they extract *Longest Common Subsequences (LCS)* from the traces to overcome junk calls, which might be a good evasion strategy. This work takes the order of function calls and their arguments into account, which can reveal extra information about the malicious intentions of the software. Lastly, Canali et al. [149] adopts a systematic approach to demonstrate how the different feature models that rely on API calls can influence the accuracy of detection. They provide a benchmark framework and present the computational limitations for different feature models such as ngram sequences and tuple-based model that only cares the order of the calls regardless of the distance between them.

"People who are really serious about software should make their own hardware."

— Alan Kay

4

Design of a Hardware Module for Runtime Attestation on Embedded Systems

This chapter presents a new attestation approach that can reason about software runtime in real time without accumulating trace information, unlike previous schemes that rely on the recording or hashing execution traces. For this purpose, it first describes a lightweight static runtime integrity model (RIM) of the software subject to attestation. Then, it presents the design of a conceptual hardware security module (HSM) that can act like a trust anchor on a critical embedded device (prover) under scrutiny. This module is designed to measure program runtime continuously through the device bus and to validate its correctness according to the given static model. Therefore, any mismatch between the model and the actual runtime can be reported when the remote party (verifier) makes a request. As the main objective, this chapter aims for a solution to avoid the main challenges (e.g., path explosion) of existing runtime attestation schemes arising from deferring and assigning all validation tasks to the verifier. Towards this goal, it mainly investigates how to offload these validation tasks to a resource-constrained hardware component, which should be already required for trace logging or hashing activities as in existing schemes. Although the chapter suggests an off-chip FPGA-based

approach without actual implementation, the same concept can also be instantiated through different architectures (i.e., on-chip extensions).

4.1 Introduction

Indispensable components of many critical infrastructures such as power grids, industrial control systems, health and transportation services, embedded devices pose significant risks because of their monitoring and control tasks that interact with the physical world. While their constrained nature—aiming for specific tasks—hinders the deployment of most security solutions available to high-end systems, the common use of unsafe languages (i.e., C/C++) for embedded software development inherits many vulnerabilities.

As a challenge-response protocol, *remote attestation* is a popular mechanism for checking the correctness of the software running on those critical systems. Remote attestation enables a remote entity, *verifier*, to ask an untrusted embedded device, *prover*, to provide integrity assurance about itself. In most attestation schemes, when the verifier makes a request, the prover calculates a cryptographic checksum of its static memory contents (i.e., code segments) and returns it to the verifier as proof. However, there are two limitations to such an approach.

The first is that occasional measurements triggered by the verifier cannot guarantee that the prover has always been in the proven state. Due to the lack of continuous monitoring, an attack scenario that starts and finishes between two attestation windows would not be caught as long as the attacker leaves the attested memory regions in an acceptable state, i.e., time-of-check-to-time-of-use (TOCTOU). Unfortunately, attempts to shorten these time gaps via more often attestation requests can have a significant impact on the availability of the device because the prover would spend most of its time on checksum calculations [68].

The second limitation of a checksum-based approach is dynamic memory regions (e.g., stack), where many runtime attacks (e.g., ROP) can be accommodated. In contrast to static code regions, the checksum of dynamic memory regions would not deliver much value to the verifier for two reasons. The first one is that the verifier

cannot reason about such a measurement unless the verifier has access to the same (external) program input with the same hardware and software settings. This is because each execution or corresponding state at a particular time would be specific to the external program input provided by the environment or the user. This makes the problem undecidable from the verifier's perspective. The second reason is that even if all external data bits are excluded from the checksum calculation, discovery of all acceptable checksum values would not still be possible for many programs, due to the combinatorial explosion in the number of states that the rest can constitute. Therefore, the returned checksum would be inconclusive again.

Many attestation schemes fail to address those issues together and ignore attacks that can exploit time gaps or memory regions left unattested. There have been attempts to address some of those limitations. For example, RATA [68] addresses TOCTOU problems by including the last modification time of the attestable (static) regions in the checksum. But this approach is not applicable to dynamic memory regions since the verifier does not know what these regions should contain or when they should be written. To reveal attacks that can touch dynamic memory regions, some attestation schemes [7, 8] record path traces on the prover and deliver them to the verifier in a lossless way, incurring storage and communication overheads. Alternatively, many runtime attestation work [5, 6, 150] provide a single cumulative hash to the verifier as a digest of trace information. Because the prover returns only a single hash value representing the whole program execution, such schemes require the verifier to discover all possible path traces and corresponding hashes in advance, using program's control-flow graph (CFG). However, this requirement overlooks potential challenges on the verifier side due to the same reasons mentioned above, which are the rapid explosion of path search space for many programs and the undecidability of the verification problem without program input in case of attacks complying with the CFG (i.e. control-flow bending).

To address these drawbacks in a more practical setting, this chapter proposes an attestation scheme that can monitor the prover's state with the help of a hardware security module (HSM) connected to its system bus. Due to continuous monitoring

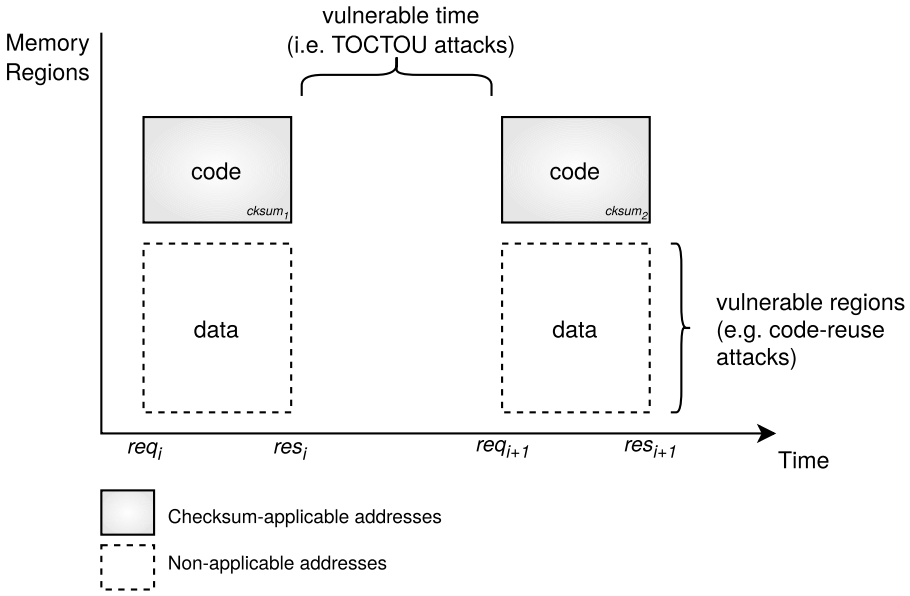


Figure 4.1: Temporal and spatial coverage limitations of conventional static attestation schemes.

of HSM, our proposal would catch TOCTOU scenarios that can temporarily alter the device software, even for a short time. More importantly, it substitutes trace-based checks on the verifier side with model-based checks to be performed in real time, for a more practical approach. Therefore, the proposed scheme does not accumulate any trace information that would incur costly storage and communication overheads, nor expects the verifier to discover path traces in advance, causing path explosion. Also, thanks to the proposal of a non-invasive hardware module design, it offers an attestation solution that can fit better into existing systems.

The rest of the chapter is organised as follows: Section 4.2 defines the problem scope and the assumptions about the system and the adversary. While Section 4.3 describes runtime integrity model (RIM) as the model that describes legitimate executions, Section 4.4 explains how the hardware security module (HSM) should use this model to check the correctness of runtime in real time. Section 4.5 explains the details of the protocol reporting on the state of the prover. Section 4.6 analyses the security of the proposed scheme. Section 4.7 approximates the resources required by HSM and its potential overheads. Section 4.8 discusses alternative implementations of the given design.

4.2 Problem Setting

Threats to the runtime integrity can be categorised into three groups as *code*, *control-* and *data* attacks, as explained in Chapter 2. The first group covers scenarios where the original program code in memory is modified or replaced by malicious code instances. Although conventional attestation schemes aim to reveal such cases, a memory measurement triggered only by the verifier's request can best prove that the code regions are in a good state by the time the request is received. As depicted in Figure 4.1, in the absence of continuous monitoring, an attacker can temporarily compromise the prover's software by altering or replacing it with malicious code, and can switch it back to the expected state prior to following the attestation request via different techniques (e.g., memory copy/hiding attacks). We refer to these time-of-check-to-time-of-use (TOCTOU) scenarios as *code attacks* in this chapter.

On the other hand, the simplest type of *control* attacks, code-injection scenarios use dynamic memory regions to load and execute custom code. Even if this is not an option, the prover can still be compromised through code-reuse attacks. An attacker crafting control data (e.g., return addresses) can maliciously reuse the original code. With a program that provides the necessary code snippets (i.e., attack gadgets), those code-reuse attacks can be Turing-complete, meaning that any (arbitrary) code can be expressed, without injecting new code or altering the existing one. For a successful *code-reuse* scenario, the attacker should exploit control-flow transfer instructions, the destination addresses of which are given from the data segments. For typical embedded software implemented using C language, the attacker would have many options: The first one is exploiting the return addresses on the stack [1, 27], known as *return-oriented programming (ROP)* attacks. Alternatively, the attacker can take advantage of indirect jump or indirect call transfers (e.g., function pointer) if the code contains [26, 28]. These scenarios are also called *jump- (JOP)* and *call-oriented programming (COP)* attacks. In this chapter, we refer to all these types as *control attacks* in general. Differently, the attacker can specifically target program variables [30, 31, 151] without touching

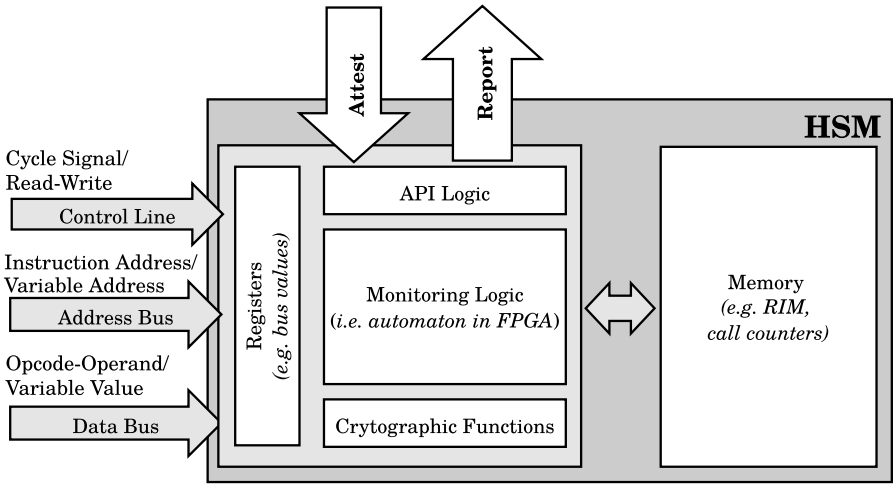


Figure 4.2: An overview of hardware security module (HSM) illustrating its internal components and external interactions.

control data, such as a global flag that can result in the execution of a privileged program path. These are also called data attacks.

4.2.1 System Model

For our scheme, we consider two entities: the *verifier* and the *prover*. As the remote party, the verifier is trusted and can ask the prover to provide a report showing that the prover is in a good state at will. We consider the prover is a simple embedded device, a single-core non-pipelined MCU without any cache. The device has single-purpose monolithic software (i.e., bare-metal). It executes instructions directly on logic hardware with physical memory addresses. The software subject to the attestation can contain indirect calls (e.g., function pointer), jumps (e.g., switch statements) and recursive functions. The design given in the following sections assumes a load-store architecture with fixed-length instructions (i.e., RISC), whose return operations rely on link register usage.

Furthermore, the prover is considered to have an off-chip memory-constrained hardware security module (HSM) that is attached to its system bus, which provides access to necessary runtime information between the CPU and the prover's memory. The HSM (as depicted in Figure 4.2) has built-in hardware implementation of runtime monitoring logic described in Section 4.4. The HSM has limited but its

own memory resources, mainly hosting a static runtime integrity model (RIM) of the program subject to attestation. This static model, described in detail in Section 4.3, is provided by the verifier and loaded into the HSM at deployment time. HSM's memory also contains some dynamic bits to keep track of the execution context and the number of calls made from each function. These bits collaborate with the static part to monitor the runtime integrity of the program. While the information on the bus provides information on how the prover's software executes, the HSM is considered to have a basic attestation API that reports the device's status to the verifier. The HSM keeps a key (sk) that never leaves its internal memory and is used to sign the attestation responses.

4.2.2 Adversary Model

Prior to the attestation, the adversary has access to the source code, binary and RIM. External resources are available to collect or record any protocol activity for later use. Only software attacks targeting memory are considered, while physical attack capabilities on both the prover device and the HSM are beyond the scope of this chapter. The adversary has the ability to write an arbitrary value to an arbitrary memory address. He can modify the program code and can revert it back to the original state (i.e., code attacks). He can also manipulate the program execution by corrupting control data (i.e., code-reuse attacks) on dynamic memory regions, though the adversary cannot affect the HSM's internal state and the verifier.

The ultimate goal of the adversary is malicious execution on the prover without being noticed by the verifier. Acting on the prover, the adversary can try to hide attack artefacts from the HSM. If this is not possible and the HSM has already detected an attack, the adversary may attempt to prevent the genuine reports from being received by the verifier and replace them with counterfeit but acceptable ones. The adversary can arbitrarily call the HSM's API to learn about the HSM's internal state or to generate signed attestation reports for later use. The adversary can intercept and modify any messages on the network or replay the responses sent earlier.

```
1  struct user_info {
2      int user_id;
3      int role_id;
4      int authenticated;
5  } user = { 0, 0, 0 };
6
7  void authenticate(void* func_ptr){
8    char user_name[10];
9    char user_pwd[10];
10   int msg_type;
11   ...
12   /*arbitrary memory write bug*/
13   ...
14   check(user_name,user_pwd);
15   if (user.role_id==2)
16         func_ptr=&priv_session;
17   ...
18   switch(msg_type){
19     ...
20   }
21 }
22
23 void login() {
24     void (*create_session)(int)=&unpriv_session;
25     while (!user.authenticated){
26         authenticate(func_ptr);
27     }
28     (*create_session)(user.user_id);
29     return;
30 }
```

Figure 4.3: Vulnerable program code that can form a basis to different attack scenarios.

4.3 Design of Runtime Integrity Model (RIM)

Prior to deployment, the verifier extracts a runtime integrity model (RIM). This model approximates the benign executions that the code can have. It is stored by the HSM internally and is used as a reference model to check whether the information captured at runtime through the system bus complies with the expected behaviour. This is a two-layered static program model centred around the call graph. The main control layer models legitimate control transfers amongst program functions such as calls and returns. The second layer enables coarse-grained checks on memory accesses, for instance, checking whether a function is allowed to access

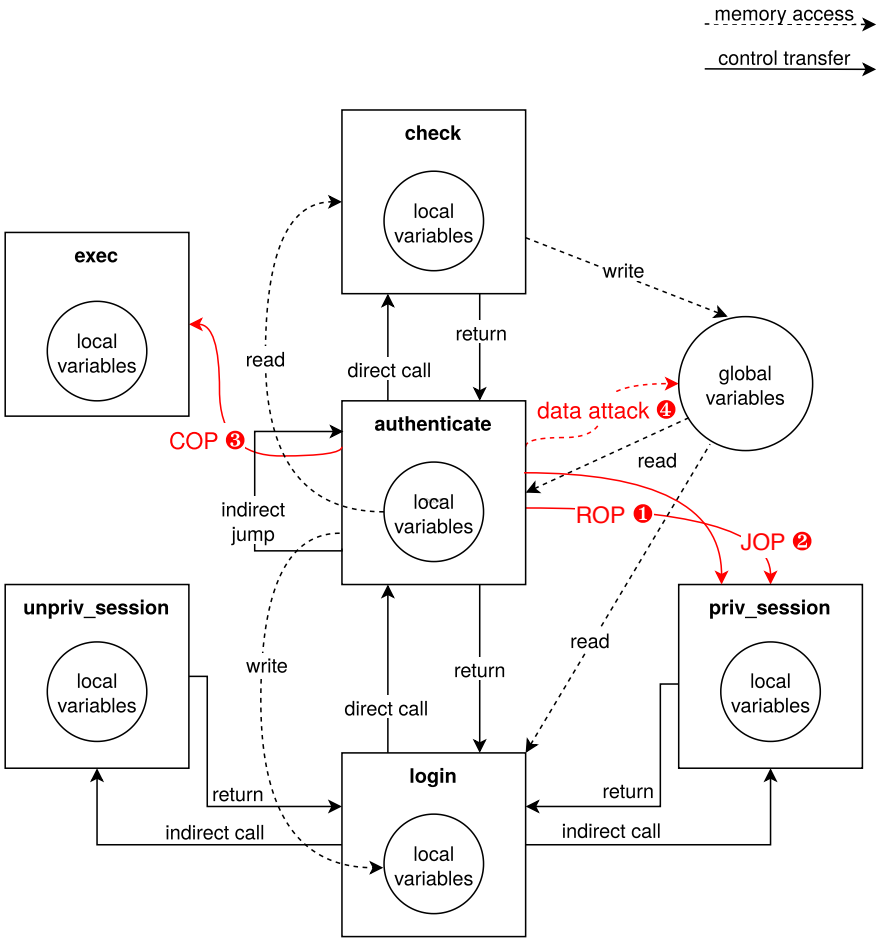


Figure 4.4: Runtime integrity model (RIM) of the vulnerable code in Figure 4.3 with potential attack scenarios illustrated.

global variables or caller frames on the stack.

To explain RIM and to exemplify the possible attacks covered in Section 4.6.3, we will use the example code in Figure 4.3. The given code assumes a vulnerable login mechanism that can form a basis for different attack scenarios. The code illustrates two functions login and authenticate using global user_info variable for login status. The former function implements the core logic and creates a user session, whereas the latter function, which checks the credentials and sets the necessary info, contains a bug that provides an arbitrary memory write capability to the attacker. Despite the details omitted, authenticate function has an indirect jump (e.g., switch statement). A corresponding RIM of this code, which is elaborated in the following sections, can be found in Figure 4.4.

4.3.1 Static Model

The RIM has different node and edge components to guide the HSM on what action is required for each instruction. As seen in Figure 4.4, nodes illustrated with squares correspond to *function blocks*, while the solid directed edges represent *control transfers* between them. These constitute the main control layer of the model. The secondary data layer is depicted by circle-shaped nodes describing local and global *data* scopes and dashed directed edges representing *memory accesses* to those.

4.3.1.1 Control Layer

The control layer has two components: function blocks and control transfers. Each function block is described as an address range consisting of the beginning and end instructions. A control transfer edge corresponds to an instruction that can change the active function (call) block. This can be a *direct call*, *indirect call* or *return* instruction, which all make the call-return graph of the program. Those edges carry address information of target instructions as permitted destinations. Although they already correspond to the beginning of function blocks for call transfers, return destinations are represented by the addresses of call sites. Additionally, for a function block that contains an *indirect jump*, e.g., switch statement, the RIM considers a self-referencing edge to the same function block unless the function that contains it is the longjmp function. This is because such instruction cannot branch outside the function in a regular scenario. If there is a non-local jump due to the setjmp/longjmp instance, the RIM adds an inter-procedural edge from the longjmp function to the function blocks that calls setjmp function. For practicality, RIM does not represent control transfers at the basic block level, such as unconditional or conditional jumps, the destinations of which are already hard-coded. These control instructions cannot be exploited without modifying the code, which would be an attack scenario that our scheme promises to detect as a code attack, as explained in Section 4.4. Although hard-coded direct calls cannot be exploited as well without code corruption first, they are represented by the RIM to keep track of the execution context at the function level.

4.3.1.2 Data Layer

RIM has an additional layer that describes a coarse-grained model of legitimate memory accesses that must be observed at runtime. This data layer consists of variable groups and memory accesses. RIM assumes that only the host function can access its local variables (i.e., call frame) unless a variable address is shared as a call-by-reference argument with a callee function. Second, despite the availability of global variables to the whole program, the code can statically describe which functions should legitimately access them.

These two layers provide a static approximation of legitimate program executions. Therefore, an attack that deviates from the expected control flow or violates given data access policies can be reported.

4.3.2 Dynamic Extensions

Validating program runtime according to a static model is inevitably subject to over-approximation limitations. More specifically, for a function that can be legitimately called from different (caller) functions, a stateless call graph does not precisely specify the exact function that the callee must return at runtime. An attacker can thus replace the return address of the original (caller) function with another function that the graph permits. A shadow stack (hosting the copy of return addresses) is typically used to differentiate such cases and achieve more precise return integrity through comparisons of shadow and actual return addresses. However, in the case of recursive functions, a shadow stack cannot be accommodated in a hardware module with limited memory resources. In this work, we extend RIM with *call counters* to attest return addresses with better precision without asking for unbounded memory resources. Each function has a counter value that is set to zero by default. This counter is incremented for a call made from and decremented for a return to that function. Since every caller would be returned in a regular scenario, those counters must be zero at the end of a legitimate program execution. The verifier can check whether these counters are compliant or not, depending on the last

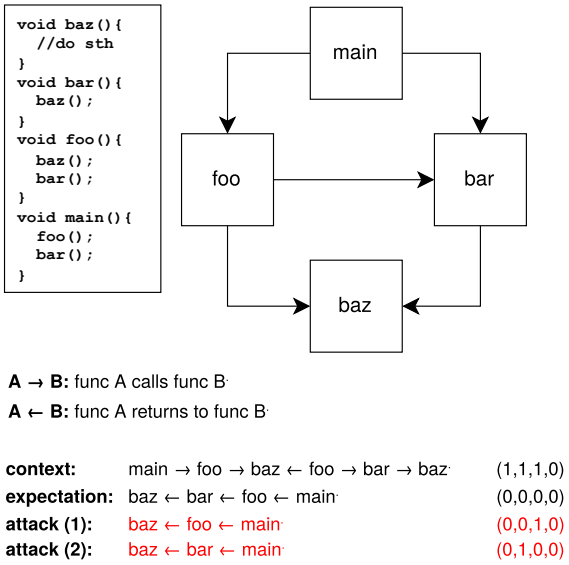


Figure 4.5: A program whose functions can be called by different callers depicts code-reuse attack scenarios that would have remained undetected without call counters.

instruction executed. Any inconsistency would reveal attacks that could have stayed unnoticed by a pure graph-based approach.

To illustrate how these counters would enhance the scheme, Figures 4.5 and 4.6 depict two synthetic examples with aligned call graphs. At the bottom of each, directed arrows represent call and return instances of their crafted traces. *context* traces describe the current call stack of each program execution, whereas *expectation* traces show how the executions should complete. Both figures provide example attack traces that pure call-graph based checks would miss. Specifically, Figure 4.5 illustrates scenarios that the attacker returns to a different function (e.g., baz←foo) that is different from the expected one (e.g., baz←bar). Figure 4.6 presents a program with indirect recursion (e.g., $\mathsf{foo}^1 \rightarrow \mathsf{bar}^1 \rightarrow \mathsf{baz}^1 \rightarrow \mathsf{foo}^2$) where the attacker can return to a different frame context on the stack, which is depicted by the numbers, while skipping some expected returns. Thanks to these call counters information, most control attack executions that would be compliant with a static call graph can be revealed.

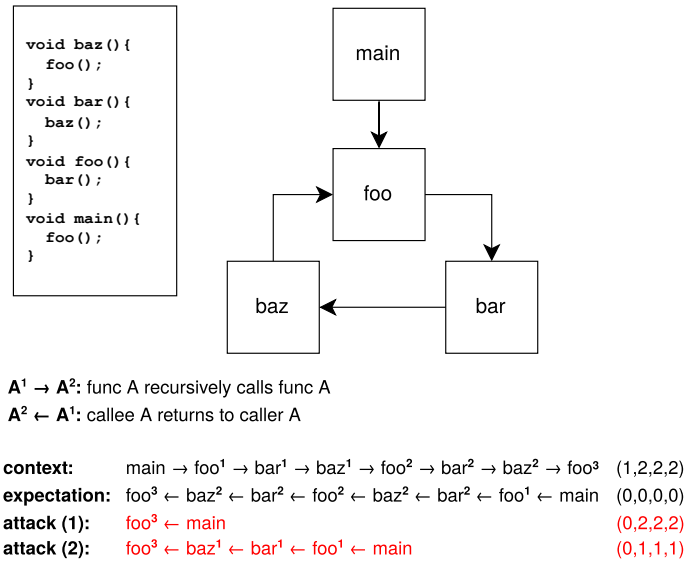


Figure 4.6: An example program whose functions constitute an indirect recursion depicts code-reuse attack scenarios that would have remained undetected without call counters.

4.4 Runtime Monitoring and Attack Detection

With the RIM loaded at deployment time, our hardware security module (HSM) connected to the prover's bus becomes ready to monitor the integrity of both program code and execution. As depicted in Figure 4.2, address bus provides instruction and variable addresses. The bi-directional data bus carries instruction (opcodes, operands) and variable (value) contents from memory, while the control bit indicates the access type. The HSM uses control and address bits to ensure that program code is not corrupted. More importantly, the address bus informs about where control transfers jump to and memory operations access to. On the other hand, data bus values are used to identify what instruction is being fetched. Using bus information as the runtime input and the RIM as a reference model, the HSM is designed to measure and check whether both code and its execution are in a good state.

4.4.1 Runtime Integrity Checks by the HSM

In order to follow up the prover's execution, HSM's monitoring logic is considered to have six different modes, each of which should complete its task in a single bus cycle (see Figure 4.7). These modes constitute a finite automaton, where each mode

corresponds to an automaton state. The HSM starts with *Dispatcher* as the default mode. Depending on the instruction on the bus, this mode causes the HSM to switch to the relevant task mode, as the name implies. These task modes are designed to monitor the compliance of control flow transfers and memory accesses with the RIM. If an integrity violation is detected, the HSM sets the appropriate attack flag and stops further monitoring. The HSM has three additional persistent modes to distinguish and report different attack scenarios. *Code attack* implies the code addresses are illegitimately accessed or corrupted. *Control attack* means a divergence from the expected call-return graph and *Data attack* indicates unexpected memory access to either global data or higher stack addresses (i.e., callers' frames). If the HSM is switched to any of these attack modes, it maintains that state and waits for an attestation request to report the attack details. The verifier needs to perform a hard reset on the HSM to restart the process in a clean state. Figure 4.8 illustrates the bus-cycle based logic of each mode, of which detailed explanations are given below:

Dispatcher. This mode first identifies the range of every address seen on the bus. For an address pointing data regions, this mode does not take any action and waits for the next bus cycle. In the case of a code address, it first confirms that the control line has a read signal as write access would mean the corruption of program code. For read access, this mode identifies the instruction type fetched. If the instruction is one of the control transfer instructions or a store/load operation that needs special treatment, it switches to the appropriate task mode. Otherwise, it maintains the same mode and waits for the next cycle.

If a call is made to a hard-coded address, the HSM switches to *Direct Call* mode. In case of an indirect call instruction, whose callee address is given by a register, HSM mode changes to *Indirect Call*. In contrast, when the instruction is a return instruction as a backward-edge control transfer, the HSM switches to *Return* mode. If the instruction is an indirect jump, the target of which is not hard-coded, the mode changes to *Indirect Jump*. The HSM does not have a special treatment for direct jumps or any conditional jump instructions since

exploiting them requires the code to be altered first. Lastly, if a memory instruction is encountered, the HSM switches to *Store/Load* mode.

Direct Call. This mode is responsible for keeping track of the execution context on the RIM graph. Following a call instruction, the address in the next bus cycle should be an entry address of a function block known by the RIM (edges). This mode gets the call address on the bus and locates it on the RIM to update the active function node. But prior to the update, it increments the call counter of the function. In addition, this mode handles setjmp and longjmp calls with special care if the target address belongs to any of these. It stores a copy of the call counters in an array structure within the HSM for a setjmp call. Later, this array of call counters are used to update the original counters if a longjmp call is encountered.

Indirect Call. This mode works very similar to the previous mode. Differently, it ensures that an indirect call such as a function pointer used is to call a permissible function target, not an arbitrary instruction or a function in the code. It first increments the call counter. Then, it checks whether the address on the bus is a defined edge by the RIM, which is also the first instruction of a permitted function. This is because we cannot allow an indirect call target to be an arbitrary instruction of the target function. Otherwise, the HSM sets the control attack flag if the model does not recognise the destination address.

Return. The HSM employs this mode to check the integrity of return addresses. When a return instruction is on the bus, this mode checks whether the target address in the next cycle belongs to one of caller sites (i.e., return edges) defined by the RIM. If not, it sets the control attack flag. Otherwise, it changes the active function context and decrements its call counter.

The call counters mentioned in Section 4.3.2 are managed by these three modes to achieve more precise return address checks. Since a stateless graph-based approach would not notice the attacker that returns to a different function, these counters aim

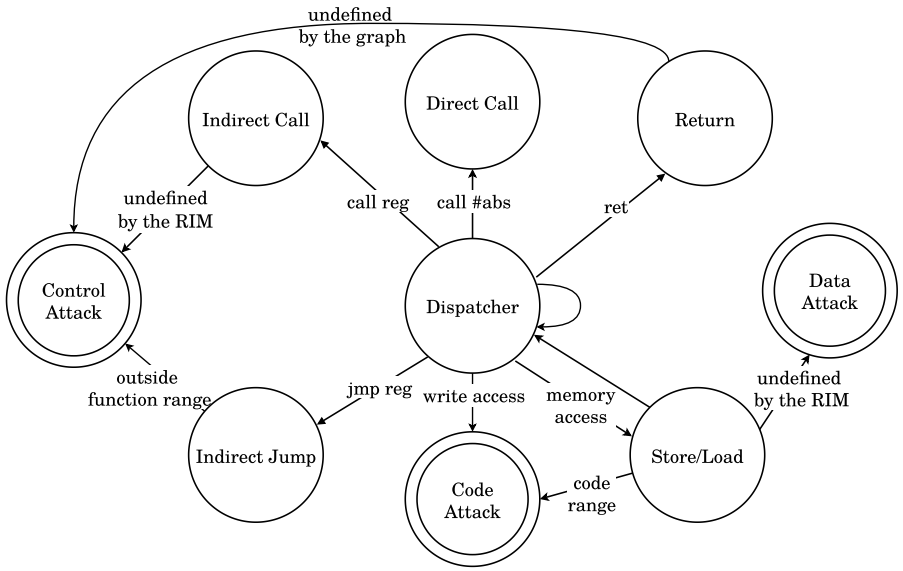


Figure 4.7: Bus-cycle based process automaton of HSM using RIM.

to approximate the shadow stack precision that would normally require unbounded memory resources in the presence of different recursive function patterns.

Indirect Jump. The HSM uses this mode to check indirect jumps (e.g., switch statement) that do not link a return. In a regular scenario, we expect jump targets to remain within the existing function. An exception to this would be indirect jumps made by the longjmp function. If the active context belongs to the longjmp function, it checks whether the target address is one of the setjmp sites defined by the RIM. Otherwise, it sets the control attack flag.

We remind the reader that direct jump instructions, both conditional and unconditional, are not monitored, as their targets are given from the code and cannot be exploited without touching the code, which is also attested by the HSM.

Store/Load. This mode performs scope-based checks to report arbitrary memory access attempts. It defines constraints on the address range in which a memory instruction can operate. First, it ensures that the program code does not write address range given by the verifier itself, which is necessary to catch code corruption scenarios. We note that legitimate self-mutating code instances are not considered. Therefore, the HSM sets the code attack flag if the operand address of a memory

instruction falls within the code range specified at deployment time. Apart from this, the mode follows up two coarse-grained policies defined by the RIM for each function block. It seeks two requirements that must be fulfilled: The first one is that a function without any global/heap variable use should not access non-stack address ranges at runtime. If such function illegitimately overwrites/reads global addresses, the HSM sets the data attack flag. Second, for a function that does not accept any call-by-reference arguments, all stack accesses must stay within the current call frame, more precisely, accesses above the current frame are described as a data attack during the execution of such a function. To perform this check, the HSM uses the active frame pointer address, which is also extracted by the mode. Because the frame pointer is also saved and restored by a store (push) and load (pop) instruction at function prologue and epilogues, this mode also keeps the copy of the frame pointer within the HSM. For this update, the mode uses the data address accessed during the frame pointer push and the data value read during the pop operation. Although the details can vary depending on the architecture and calling convention in use, it takes the offset of any non-register arguments into account.

4.4.2 Attacks Coverage

The HSM's monitoring logic is designed to reveal different attack classes: The first class is the attacks that corrupt the original program code. Thanks to *Dispatcher* and *Store/Load* modes, the HSM describes any overwrite of the given code address range via a memory instruction from that range as a *code attack*. This provides strong code integrity attestation for embedded systems that lack architectural support for code and data separation, i.e., write-xor-execute (W⊕X). In addition, thanks to continuous monitoring, it promises to capture TOCTOU attacks that would normally have remained unnoticed between two attestation windows.

The second attack class covered is *control* attacks such as *code-reuse* and less sophisticated *code-injection* scenarios, where the primary target of the attacker is control data, such as code pointers. The HSM confirms that any instruction updating the program counter with a potentially corrupted value sets the counter

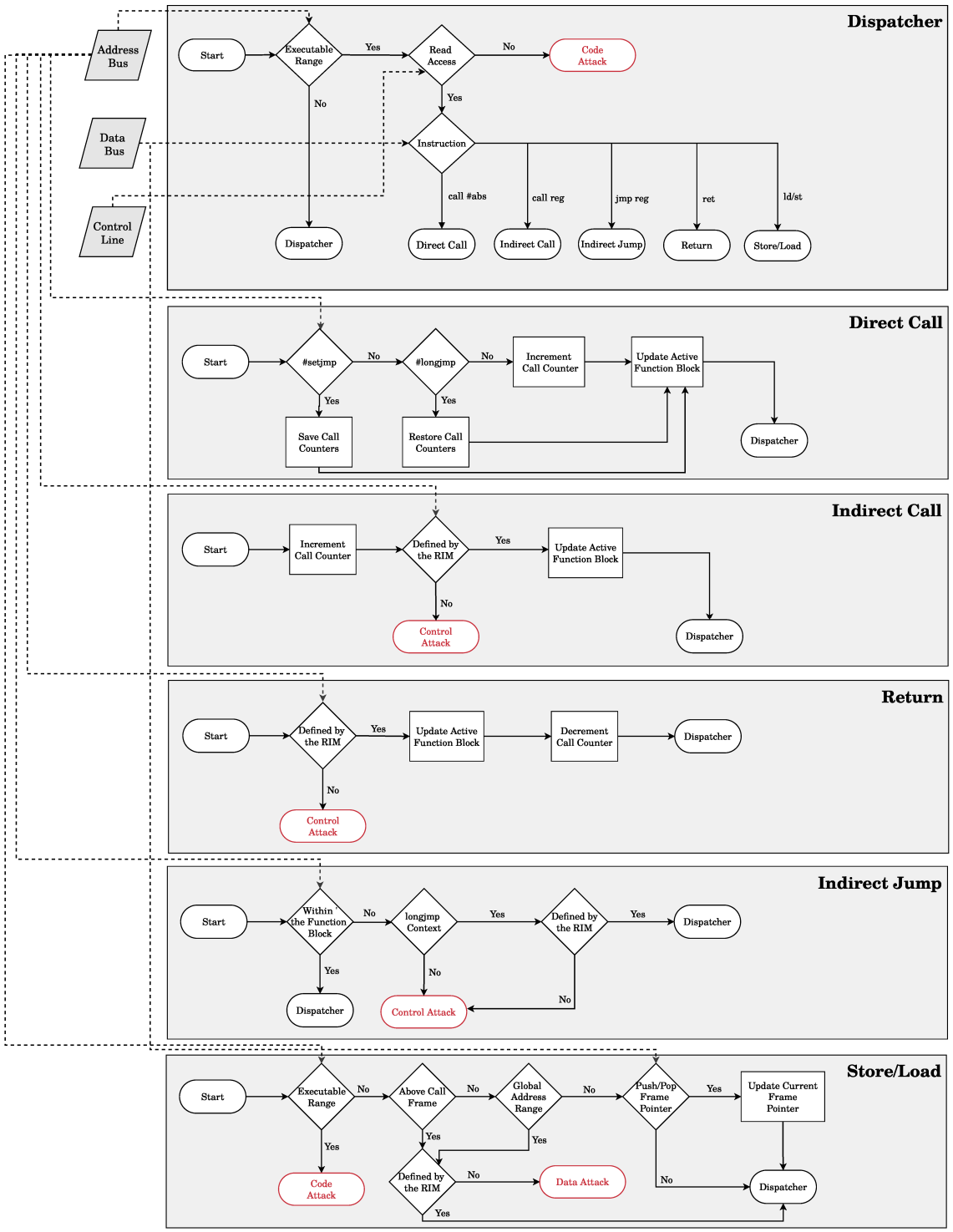


Figure 4.8: Detailed process flow of HSM's runtime monitoring logic that check RIM compliance at each bus cycle.

to a permissible instruction defined by the RIM. This includes both backward-edge return addresses (ROP) and forward-edge targets such as indirect call (COP) addresses. We do not worry about direct conditional and unconditional jumps since their destination addresses are hard-coded and cannot be altered without a code attack first. To start executing an injected code from non-code address range or to reuse already existing instructions, the attacker must take over at least a single code pointer. This should eventually cause a divergence from the RIM and will be captured by the HSM as a control attack. For a better reduction of the attack surface, call counters reveal side cases where the attacker crafts return addresses with options that do not diverge from the call-return graph. Despite not being as precise as shadow stacks that preserve the order of calls, call counters significantly reduce the options for the attacker that would be given by a stateless graph. The HSM also checks the constraints described by the RIM for both intraprocedural and interprocedural indirect jumps to reduce usable attack gadgets.

In addition, our scheme considers *data attacks* that reuse the code without altering code pointers. We remind the reader that complete coverage of data attacks normally requires either memory safety or expensive fine-grained data-flow integrity checks, which is a non-trivial task to perform with HSM's limited resources. Therefore, the HSM offers only coarse-grained checks. These checks aim to catch accesses to global data by a function without any expected use or accesses to the callers' frames by a function that does not take any reference/pointer arguments. Memory accesses that do not comply with those constraints would thus be reported as a data attack.

4.5 Protocol Overview

This section presents a protocol design that assures the verifier receives a genuine report through an infected device and untrusted network. We consider that, at any moment, the verifier can make a request to learn about the prover's internal state. As seen in Figure 4.9, when the prover receives an attestation request containing a fresh nonce value N generated by the verifier, the prover calls the

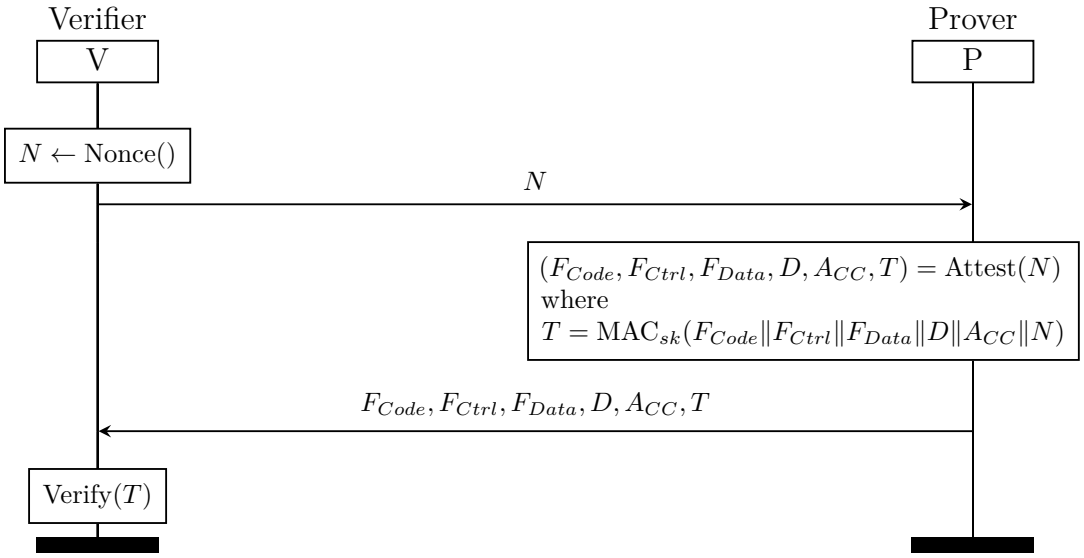


Figure 4.9: Overview of the remote attestation protocol reporting any attack flags and auxiliary information to the verifier.

Attest(N) provided by the HSM's API. Then, the prover needs to send back its output as the attestation response to the verifier. The response consists of code F_{Code}, control F_{Ctrl}, data F_{Data} attack flags, diagnosis information D about the state prior to the attack, which consists of two registers holding the last executed instruction address and the destination address attempted by a control or memory instruction and the array of call counters A_{CC}. We remind the reader that HSM stops further monitoring in case of an attack flag is set. Therefore, the response provides information to the verifier to reason about the instruction exploited and the intended target. Additionally, each response contains a tag T of which all the information and the sent nonce are signed with a MAC scheme. Upon receiving the attestation response, the verifier verifies the tag using the shared key with the HSM. While the key (sk) ensures the authenticity of the message, the tag—digesting nonce N, flags and counters—guarantees the freshness and integrity of the response. The verifier can then check flags and counters to decide about the existence of an attack. F_{Code} flag means a code attack. F_{Ctrl} and F_{Data} flags imply a runtime attack scenario, where the former states a control hijack while the latter tells us there is a data access that violates the policies stated by the RIM. Only if all

flags are negative and call counters are zero/compliant as expected, the verifier can conclude the prover is in a healthy state.

4.6 Security Analysis

To successfully compromise the prover without being detected by the verifier, the adversary must either hide the attack artefacts from the HSM or forge a valid attestation response when queried by the verifier. This section analyses these possibilities, with an evaluation of the attacks captured by the RIM on a concrete example.

4.6.1 HSM Attacks

Due to the bus integration, every instruction executed and data transferred from/to memory will be monitored. Because physical attacks (e.g., probing) are excluded, any attack has to go through the bus and will be accessible by the HSM. The adversary should modify either the code or its control flow for an attack. Alternatively, the attacker can attempt to find a flaw in HSM's monitoring logic.

Regarding the first option, if the adversary uses a memory instruction from the given range to modify the code itself, HSM will report those as code attacks thanks to *Store/Load* mode. Hijacking control flow as a code-injection or -reuse attack is also not practical since the attacker's execution must comply with the RIM. But RIM put constraints on all control instructions, the target addresses of which might reside on dynamic memory regions. *Indirect Call*, *Return* and *Indirect Jump* modes guarantee that the program counter is always set to an instruction address described by the static model. Additional call counters cover scenarios that might exploit the imperfections of the static return edges. Considering the constraints defined by *Store/Load* mode, the attacker's ability to manipulate control flow via data attacks is also reduced. As a result, the HSM would catch the attacker for a scenario that does not comply with the RIM.

Regardless of the compliance with the RIM, for an attack targeting HSM's monitoring logic, the adversary must find a bug/flaw that can alter the RIM or

dynamic states within the HSM. However, this is unlikely because the monitoring logic implemented as hardware would be free from software vulnerabilities providing too much scope to the attacker with arbitrary read/write capabilities.

4.6.2 Protocol Attacks

When altering the HSM states is not possible, the only option left to the adversary is to prevent the verifier from seeing the genuine violation flags. To accomplish this, the adversary has to return a valid response to the verifier's request. If the prover does not respond, the verifier will conclude the prover is compromised. Therefore, the adversary cannot simply block a message or remain silent after compromising the prover. There are only two ways an adversary can send a valid response: either replay a previously captured response or craft one from scratch. We look at each of these in turn.

MAC provided with the response (Figure 4.9) contains a nonce picked by the verifier. Thus, to replay the response message, the adversary would either have to force the verifier to use the same nonce twice or predict what nonce is going to be used and query the prover ahead of time before compromising the prover to obtain a clean response. This is only possible with negligible probability since we do not allow the adversary to compromise the verifier, and the nonce is chosen securely (i.e., uniformly from a large domain).

Thus, to return a valid message, the adversary must create it from scratch. However, the message must be authenticated using a key kept in the HSM, which the adversary cannot obtain by assumption. Therefore, to forge the message, the attacker has to break the existential unforgeability property of the underlying MAC scheme, which can be done only with negligible probability.

4.6.3 A Concrete Example

This section discusses the effectiveness and limitations of our scheme against attack scenarios that could be performed using the code in Figure 4.3. A powerful attacker exploiting the arbitrary memory write primitive given in *line 12* would have different

options: For example, as a control attack, he can replace the awaiting return address on the stack, which should normally point to the call site at *line 26*, with the address of a different function ❶ such as priv_session. Or he can alter the target address of the indirect jump generated by the switch statement in *lines 18-20* to perform a jump to any instruction, such as *line 28* or the priv_session function ❷ as a desired outcome. Alternatively, he can corrupt the function pointer defined at *line 24* with the address of a critical system function ❸ (e.g., exec). Also, he can modify global user_info elements defined at *lines 5* as a data attack example ❹ that would help to create a privileged session without any legitimate authentication. For any of these scenarios, the HSM would set the corresponding attack flag, as they all constitute a deviation from the RIM graph depicted in Figure 4.4.

In terms of limitations, we note that RIM cannot approximate all legitimate executions with full precision, like any static model. For example, if the attacker replaces the address of unpriv_session with the address of priv_session, the HSM would have to give a pass to such a scenario, as both are valid targets according to the model. Or our scheme does not have much to do if the attacker performs a meaningful attack by exploiting the indirect jump of authenticate while staying within the range of the function. Identifying such attack scenarios is not possible without the knowledge of program input, which is a known limitation for any static approach. For completeness, we also highlight that our coarse-grained checks on memory accesses leave room for a data attack scenario targeting stack variables at higher frames from a function that has at least a single call-by-reference argument or an attack targeting another local variable within a function. Detection of such data attacks requires more fine-grained checks such as DFI [4], which cannot be accommodated in a hardware module with very limited resources.

4.7 Performance

The HSM is designed to operate in real time while the prover keeps running. We remind the reader that the prover has a general-purpose CPU and enough memory resources to host call frames of recursive functions. In contrast, the

Table 4.1: Relevant metrics extracted from different bare-metal examples for the approximation of size and search complexities of runtime integrity models (RIM).

	Instructions			**Functions**	**Highlights Per Function**			
	Count	Call[%]	Ret[%]	Count	Avg-Call	Avg-Ret	Max-Call	Max-Ret
jtag	165	12.1	1.8	5	0.8	3.3	3	10
bootldr	554	12.8	2.7	17	1.3	3.4	6	8
zlib	9068	2.6	1.3	57	1.4	2.7	8	19

HSM has limited memory and serves a specific purpose, for which its hardware is tailored. For a practical attestation scheme, both the memory usage and the complexity of the HSM tasks should comply with its resource constraints without degrading security guarantees.

In terms of memory requirements, the HSM must provide enough space to host the RIM and call counters. The size of the RIM can be defined as $\mathrm{O}\big(n + e\big)$, where n is the number of nodes and e is the number of edges in the model. The former corresponds to the number of functions, whereas the latter is mainly defined by the number of call edges from a caller function to distinct functions and the number of return edges to different call sites. We note that both (intraprocedural) indirect jump and memory access edges illustrated in Figure 4.4 do not scale up per function nor increase the complexity of the RIM since their checks are not address-specific. Hence, the number of functions and call-return relations between them represent the main cost of the static model. For the dynamic part, the space required by call counters is also defined by the number of program functions, regardless of the depth or recursiveness the call stack might have at runtime. Despite being program-specific, we can approximate the memory requirement of a RIM as a function of the program size. To provide insight into such an evaluation, we have analysed three bare-metal examples of different sizes. Those binaries are JTAG, bootloader and compression library implementations with components including UART, Adler, CRC32 checksums and memory allocators. Table 4.1 summarises the number of instructions and key RIM components found in those instances. We highlight that RIM, centred around the program's call-return graph, provides a

more succinct representation of the binaries with smaller sizes. For instance, zlib, as the most complex example consisting of more than 9K instructions, is modelled using a far less number of components with 57 function blocks (address ranges) and an average of 1.4 call and 2.8 return (address) edges per function. With an average of 14% model size to binary size ratio, RIM requires reasonable memory resources.

Regarding the complexity of HSM tasks, each depicted mode has a different process flow. Many modes such as *Dispatcher*, *Indirect Jump*, and *Store/Load* fulfil their tasks within constant time. On the other hand, *Direct Call*, *Indirect Call* and *Return* modes perform a linear search task whose cost is normally defined by the degree of active function node in the RIM. However, those searches are expected to be bounded in practice due to the limited number of functions. For instance, *zlib*, as the most complex bare-metal instance examined, does not include any function block with more than eight call edges and less than 19 backward edges, as shown in Figure A.3. Similar to previous attestation proposals [6, 150], those searches can be parallelised using a content-addressable memory (CAM) buffer that would host the data of the active function block and complete the search in constant time. We highlight that each mode aim to complete its task within the same bus cycle. To perform these tasks in real time, we consider a non-generic hardware-based implementation such as FPGA for the monitoring logic described in Figure 4.8. Implementing the monitoring logic at a lower abstraction layer would allow HSM to process much faster. The entire process of each mode could be completed in a single tick of the FPGA's clock, as there is no data dependency that would block the completion of given HSM tasks (modes) within that bus cycle.

4.8 Discussion

Despite the off-chip proposal in this chapter, we highlight that our monitoring logic, as the core contribution, can be instantiated through different architectures. For example, recent VRased architecture [65, 67, 68] extend OpenMSP430 with an on-chip hardware module (HW-Mod) for attestation purposes. This module extracts similar information directly from the MCU core instead of the device bus, and

generates control-flow traces for the verifier [8]. Extending VRased to implement our monitoring logic instead of a trace-based approach would be an interesting direction to explore. Alternatively, similar to the recent work DExIE [152] that suggests enforcing control flow integrity as part of pipelining, our monitoring logic can be adapted to the core. Or customisable microcontroller platforms (e.g., discontinued AT91CAP7 series) that can provide native MCU-and-FPGA integration via metal programmable cell fabric [153] and faster FPGA interfaces [154] can be considered as other options to instantiate our runtime monitoring logic. Due to economies of scale, such devices with more promising costs (e.g., $6-13 [155]) in large quantities could be used to target even non-critical domains, where the additional cost of a module might be more difficult to justify.

Although each approach has different use cases, validating the compliance of runtime based on a static model using minimum resources and hardware-implemented logic poses similar research challenges, which makes this chapter and its contributions relevant to different architectural adaptations.

4.9 Summary

This chapter presented a novel attestation approach to report attack scenarios that can violate the code and execution integrity of critical embedded systems. This approach is based on a conceptual hardware module (HSM) loaded with a static lightweight model (RIM) of the program. Unlike conventional attestation schemes, our approach can capture TOCTOU scenarios that exploit temporal gaps between two verifier-triggered measurements for code replacement and corruption.

In addition to code attestation, our scheme offers an attestation mechanism that measures how the legitimate code is executed according to the reference model (RIM). Therefore, without having to accumulate runtime traces, which incur significant storage and communication overheads, or without having to digest them into a single hash, which makes the verifier subject to path explosion problem, our approach can reveal both code and code-reuse attacks to the verifier. As the main components of the scheme, both runtime integrity model and the monitoring

logic are designed in a way that the attestation can be achieved using a memory-constrained hardware module.

"Trust dies but mistrust blossoms."

— Sophocles

5 Identifying Critical Variables for Lightweight Runtime Protection

With a focus on control flow, the previous chapter suggested only coarse-grained checks against data attacks, as it is difficult to provide complete assurance (i.e., data-flow integrity) against those attacks using a resource-constrained hardware module. This chapter explores this issue further in a non-attestation context and presents a lightweight approach that can address data-oriented attacks with less resources and overheads. For this purpose, it describes a novel method to identify and check data variables that are critical to runtime correctness. Instead of inspecting every variable or memory access, our method is concerned only with the ones whose values are defined by trusted agents (e.g., the programmer), and ignores other non-critical variables that are unlikely to be targeted by the attacker.

5.1 Introduction

Despite more than three decades of effort, memory bugs are still unsolved and stay as the mother of all evils for computer security. Applications developed in unsafe system programming languages such as C and C++ inevitably host many of these bugs as their complexity and lines of code (LOCs) increase. High-level languages (e.g., Java) aim to solve the issue by assuring memory safety at runtime.

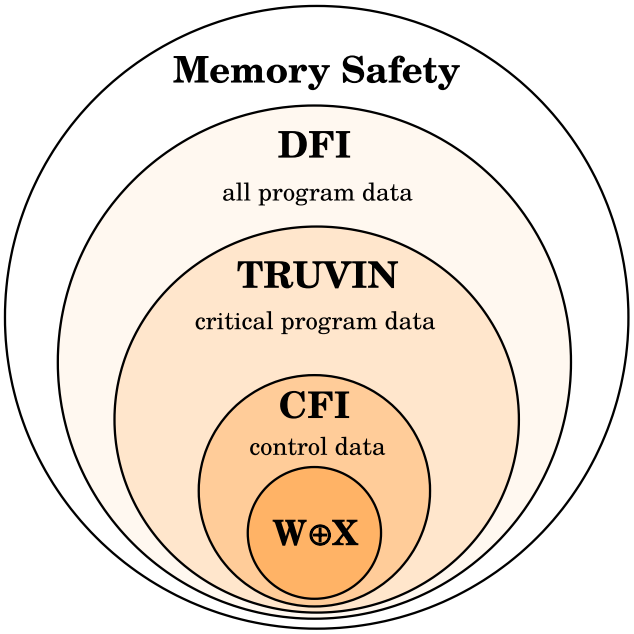


Figure 5.1: Coverage of different memory protection schemes.

However, they are not able to fully replace those unsafe programming languages due to their performance penalties and inevitable dependencies on systems that are also developed in unsafe languages (e.g., operating systems, JVM). Even if there are attempts to make those languages safer, such as bounds checking [83, 156], they are often subject to similar performance costs.

Memory bugs (e.g., buffer overflow) can form a basis for *control-* or *data-oriented* exploits. A control attack hijacks the program's control-flow by taking over a code pointer such as a return or an indirect jump address. Control-flow protections [2, 3] (e.g., CFI), provided as a feature by modern compilers today, can effectively mitigate control attacks. Despite their reasonable overheads (ca. 15%), these protection methods do not cover data-oriented attacks, where the adversary corrupts (non-control) program data without touching any code pointers. Data attacks can enable the adversary to reach his goal in a different way, for example, by altering condition variables that decide on the branch decisions.

A potential solution to data attacks needs a better approximation of memory safety, as pictured in Figure 5.1. Such an approximation typically needs to check the compliance of memory accesses with static data (flow) features of

the program, i.e., DFI [4]. Unfortunately, a software-based scheme can incur impractical performance overheads (c.a. 104%) due to the checks on almost every memory operation, while hardware-based schemes (HDFI [9]) suffer from substantial deployment costs. For a lightweight and practical solution, we need targeted approach that protects only program data critical to the runtime integrity. However, this is a nontrivial task; because deciding on the criticality of a variable needs a semantic understanding of the program source code, when not annotated by the programmer. Oversimplifying the problem as protection of all condition variables [71] would be incomplete, since there might be other variables directly used by sensitive functions. In addition, some condition variables can be legitimately defined by the user or environment, where integrity checks would be unnecessary. Despite some targeted DFI proposals that use domain-specific knowledge (e.g., kernel error codes [13]) or programmer annotations [14], there is a lack of a generic approach for user programs. This chapter aims to address this problem by suggesting a new approach to identify critical program variables, without having to fully understand the program semantic.

This chapter presents TRUVIN, a lightweight software-based scheme to identify critical program variables and to enable their protection selectively against the exploits of memory bugs found in C-like programs. In principle, our scheme describes program variables with values originating from trusted agents (e.g., the programmer) as critical, in order to avoid redundant instrumentation or checks of non-critical ones which are already controllable by agents that are potentially malicious.

The rest of the chapter is organised as follows. Section 5.2 explains our motivation, system and threat models. Sections 5.3 and 5.4 present the design of TRUVIN. Section 5.5 provides details of proof-of-concept implementation on a concrete example, whereas Section 5.6 discusses performance gains and security promises.

5.2 Problem Setting

Software-based data-flow integrity (DFI) [4] protections could not be widely adopted, mainly due to high performance overheads. In contrast, hardware-assisted solu-

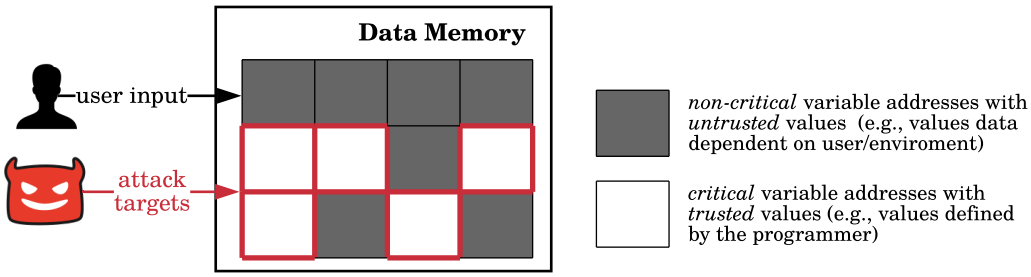


Figure 5.2: Separation of variables as critical and non-critical based on trustworthiness of their value agents.

tions [9] require expensive CPU changes despite their low overheads. Therefore, a targeted software-based technique would be practical if instrumentation overheads had been reduced. However, choosing the program data in need of primary protection is a challenging task, which this chapter aims to address.

5.2.1 Motivation

For a DFI scheme [4] relying on *reaching definitions* analysis, performance overhead is mainly defined by two factors. The first major one is instrumented memory operations, most of which correspond to variable *definitions* or *uses*, and can be roughly approximated from the number of program variables. The second minor factor is the cardinality of *reaching definitions* sets that determines the search cost for each memory read. In order to minimise the cost arising from both factors, this chapter adopts a novel targeted approach that addresses data-oriented attacks with far less overhead.

Since the overhead is assumed to be proportional to the number of memory operations, our approach selectively checks the integrity of only variable values, which are expected to be the primary attack objectives while avoiding redundant instrumentation on others. To achieve this, at a broad level, our scheme classifies variables into two groups as *critical* and *non-critical* based on the trustworthiness of agents contributing to their values (see Figure 5.2). We formulate critical variables as addresses hosting *trusted* values that originate from reliable agents such as the programmer, for the given program point in a flow-sensitive setting. On the other hand, non-critical variables are described as addresses containing *untrusted*

values that are directly or indirectly (i.e., data dependencies) defined by potentially malicious agents such as users or the environment.

Because non-critical targets can be controlled even by legitimate users, the adversary cannot benefit much from corrupting their values. For a typical data-oriented attack, the adversary needs to overwrite some critical variable value in a way that the legitimate program semantic (i.e., the programmer) does not anticipate. Accordingly, untrusted agents (e.g., users) as potentially malicious agents must not overwrite critical values directly or through data dependencies, and must not have an impact on those beyond the legitimate semantic, i.e., control dependencies only. Based on these insights into program variables, to minimise the instrumentation overhead, we can avoid checking the integrity of untrusted values hosted by non-critical variables, which are expected to be only preliminary objectives, but not the ultimate attack targets.

5.2.2 System and Adversary Model

We consider the system to have code integrity, data execution prevention (W⊕X), and control-flow [2, 3] protection. The system secures the instrumentation data (i.e., shadow memory) through randomisation or other means such as (e.g., TEE). We do not make any assumptions about how this is achieved. Although the program code can contain memory bugs, the programmer's intentions are correct, which means the program is free of logical or semantic flaws. We assume that the program enables us to precisely identify its variables via their corresponding instructions (i.e., address operands, alias sets), and consider a flow-sensitive pointer analysis that can accompany our approach. Orthogonal research problems such as inevitable limitations of static approximations or other precision issues due to imperfections of points-to analyses are not within the scope of this chapter.

The adversary's goal is to modify any critical variable, i.e., a variable that holds a trusted value, typically given by the programmer at compile-time. Trusted values do not directly depend on any input that is potentially malicious, but are instead mostly controlled by the program semantic to reflect changes in the internal state. In

practice, an attacker could, for example, modify such a variable value by overflowing non-critical variable addresses in the same stack frame. But we do not make any assumptions about how corruption is achieved. The adversary has full control over the value of all non-critical variables of the program. However, the adversary cannot interfere with the instrumentation process, meaning that he cannot modify the instrumented binary and shadow memory allocated to host the instrumentation data. The attacker will also fail if any control data is disturbed, given that the system assumes a perfect control-flow protection in place. The program does not provide a pure information leakage scenario, where only confidentiality is compromised without harming the data integrity.

This model captures data-oriented attacks extensively, including those corrupting only a specific variable or more sophisticated Turing-complete DOP attacks. It helps us not only to focus on lightweight mitigation of those attacks but also lets other orthogonal problems' solutions (e.g., isolation) be adaptable to our scheme.

5.3 Distinguishing Variables with Trusted Values

Program data should be derived from either external agents or through internal program semantic. The semantic can bring two types of dependencies on external data. The first true dependency is data dependency, where the outcome of an operation is directly dependent on the operands of that instruction, e.g., the result of an arithmetic operation is dependent on the operands. The second is control dependency, which can be described as a situation where the execution of an operation is conditional on the features of dependee data, e.g., a control structure that compares a dependee variable against some value.

Therefore, our targeted approach in this chapter leverages the fact that *program data is control or data dependent on each other, the value origins of which have to be either trusted or untrusted, while describing control dependencies as the only legitimate interface to reflect trusted values from untrusted input.* Figure 5.3 depicts this core insight, which our work counts on. By identifying critical variables based on the trustworthiness their value origins, this chapter first enables separation

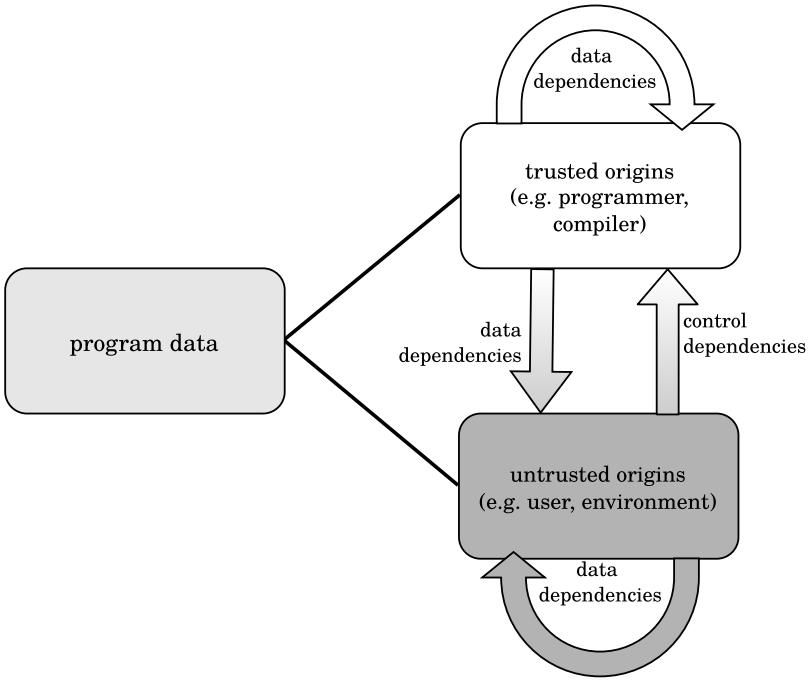


Figure 5.3: Identification and propagation of trusted (critical) program data based on value origins and program dependencies.

(isolation) of critical data from non-critical data. Then, it places fine-grained integrity checks among the critical variables. For this distinction, we first perform a novel static analysis on the program to identify critical variables that host trusted values, which can be only control dependent on untrusted values.

5.3.1 Trust Sources and Propagation

Analysing the trustworthiness of variable values is two-fold: trust sources and trust propagation. We formulate *trust sources* as origins that hold integrity guarantees and assign their values via trustworthy processes such as variable (re)definitions by the programmer. Because of the immutability of the code (where programmer values/constants are stored) and basic assignment operation (undoubtedly free of bugs), programmer-defined values introduce trust to the system. The analysis module also allows one to define other trust sources. For example, the analyst can define external trust sources (e.g., library functions), which are not part of the program code. Also, the assignments from specific database or configuration

files—considered to have integrity properties—and custom functions placed for sanitisation can be stated by the analyst.

Apart from the sources that introduce trust, the analysis also propagates trust through program statements to discover other emerging trusted variables, or the ones of which trust vanishes. *Propagation* rules are defined conservatively for different statement types as below:

- *Assignment*: If a trusted value is copied to another variable, the assigned variable value is also trusted (e.g., $t_2 = t_1$).
- *Unary Operation*: If the operand of a unary operation is trusted, the result is also trusted (e.g., $t_2 = -t_1$).
- *Binary Operation*: If both operands of a binary operation are trusted, the result is also trusted (e.g., $t_3 = t_1 + t_2$).
- *Compare*: If at least one of the comparison operands is trusted, the result is also trusted (e.g., $t_3 = (u_1 < t_2)$). (i.e., this exceptional rule enables the programmer to use control dependencies as the interface/sanitisation mechanism to reflect trust from untrusted values.)
- *Call*: If the function is a trust source, the function output (arguments/return) is also trusted (e.g., $t_1 = f()$).
  If the function is a sanitising one, specified output is trusted (e.g. $t_2 = f(u_1)$).
  If the function is a trust propagating one, based on the fulfilment of the argument of the propagation rule, the specified output is trusted (e.g., a function returning trusted value in case the first argument is trusted: $t_3 = f(t_1, u_2)$).

While being *untrusted* corresponds to the negation of all above; we deduce the following from the given formulations and rules in a flow-sensitive setting:

1. a variable whose value is defined from the code-segment (i.e., programmer-given) is trusted unless it is overwritten or poisoned by untrusted values.

```
Function analyseFunction(F)
    Result: in, out sets
    CFG ←getTopoSort(F);
    worklist ← {CFG[0]};
    while worklist ≠ ∅ do
        n ← worklist.pop();
        out' ← out[n];
        foreach p ∈ pred[n] do
            in[n] ← ⋂ out[p];
        end
        visitInstNodeAndUpdateSets(n);

        out[n] ← gen[n]∪(in[n]−kill[n]);

        if out' ≠ out[n] then
            foreach s ∈ succ[n] do
                if s ∉ worklist then
                    worklist.push(s);
                end
            end
        end
    end
```

(a) Function-wise trust propagation.

```
Function analyseProgram(P)
    Result: trust sources, prop rules
    CallGraph ← getRevTopoSort(P);
    foreach F ∈ CallGraph do
        analyseFunction(F);
        if F always returns trusted then
            addAsTrustSource(F);
        end
        foreach arg in F do
            in[entry][arg] ← trusted;
            analyseFunction(F);
            if F always returns trusted
             then
                addPropagationRule(F,arg);

            end
        end
    end
```

(b) Extraction of call-based propagation rules.

Figure 5.4: Algorithms responsible for program-wide static trust propagation analysis.

2. a variable whose value is defined by a trusted function (i.e., a function always returning programmer-given values or stated as trusted/sanitising) is trusted unless it is overwritten or poisoned by untrusted values.

3. a variable whose value is defined by a propagating function and has a trusted argument complying with one of the propagation rules is trusted unless it is overwritten or poisoned by untrusted values.

4. a variable of which all data dependencies are on trusted values is also trusted unless it is overwritten or poisoned by untrusted values.

5.3.2 Static Trust Analysis

To identify critical variables hosting trusted values according to the rules above, we adopt a static flow-sensitive approach that uses the iterative data-flow analysis framework [157], which is widely used to solve instances of data-flow problems. The iterative framework first sets up data-flow equations for the control flow graph

(CFG) nodes to represent their entry (in) and exit (out) states. Then, it solves them at compile-time by repeatedly performing the abstract interpretation of program statements until the whole system converges (stabilises). Abstract interpretation is often defined by three parameters. The first is *transfer function* which simulates the execution of the instruction(s) of each CFG node n. The second parameter is the *direction* of the analysis (i.e., forward or backward). And the last one is *join operator*, which can be either union (may) or intersection (must), stating how to combine the property flowing from predecessor or successor nodes.

Since our protection requires fine-grained information about the memory operations to be instrumented, for our analysis, each CFG node n corresponds to a single instruction rather than using basic blocks. We formulate our transfer function as follows:

$$\mathrm{f}(n) = gen[n] \cup (in[n] - kill[n]) \tag{5.1}$$

where;

$gen[n]$ is the set of introduced trusted values by the instruction in node n,

$kill[n]$ is the set of values whose trust vanished by the instruction in node n,

$in[n]$ is the set of trusted values at the entry of node n,

Because the trust originates from earlier statements (e.g., trust sources), it is a property that flows *forward*, which gives the direction of our analysis. We choose *intersection* as the join operator due to the exclusive definition of trust which is inherently conservative. As a *forward must* analysis, our data flow equations become:

$$\mathrm{in}[n] = \bigcap_{p \in pred[n]} out[p] \tag{5.2}$$

$$\mathrm{out}[n] = f(n) \tag{5.3}$$

where;

$out[n]$ is the set of trusted values at the exit of node n (p means predecessor).

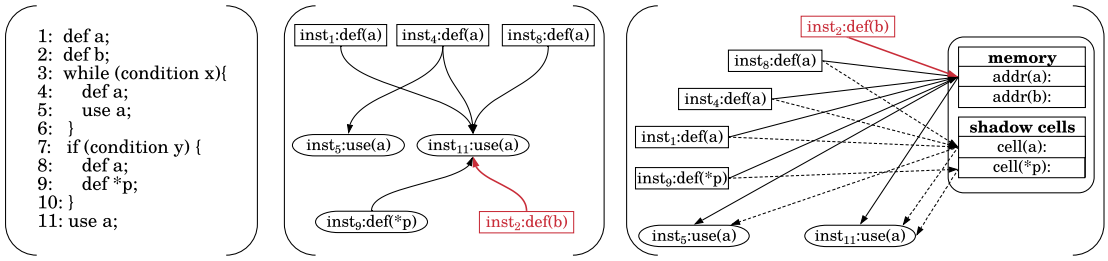
For the outcomes of Equations (5.2) and (5.3) to be converged, we have used function-wise worklist algorithm given in Figure 5.4a to solve the equations at function level.

For a program-wide analysis, this algorithm is applied to each function in the reverse topological order of the call graph (i.e., bottom-up traversal). Therefore, a caller function can use the result of the analysis of the called function for a scalable interprocedural trust analysis. Functions with arguments run this algorithm multiple times to evaluate the trust propagation of each argument to the function output separately (assuming that the argument of interest holds a trusted value). Based on discovered patterns on the function output such as the return values and arguments called by reference, propagation rules are continuously created during the bottom-up traversal of the call-graph. These rules describe which function arguments can propagate its trust or whether the function can act as a trust source even if none of the argument values is trusted. The created rules enable the abstract interpretation of call statements for an interprocedural analysis. as seen in Figure 5.4b. Upon completion of the bottom-up traversal of the call graph, the program-wide analysis provides stable entry (in) and exit (out) states for each program instruction. These sets inform us about the trustworthiness of variable values on the given instruction/node. This information already encloses memory instructions that must operate with trusted values, which corresponds to the critical variable instructions.

Despite its interprocedural approach, the analysis is deliberately designed as context-insensitive. Because the instrumentation code of a function should not differentiate for different calling contexts, we have not considered a context-sensitive trust propagation that would not benefit the transformation phase.

5.4 Detection of Data-Oriented Attacks

After instructions operating with trusted values are identified as critical, the program is transformed in such a way that attacks modifying those can be detected.



(a) example defs/uses. **(b)** DFI via reaching definitions. **(c)** value integrity via shadow cells.

Figure 5.5: Comparison of DFI relying on reaching definitions and the value integrity scheme using shadow cells.

5.4.1 Value-Based Integrity Checks

DFI scheme proposed by Miguel et al. [4] is a pioneering technique to mitigate data-oriented attacks. But this naive approach is not suitable for targeted protection only specific variables because it has to record all memory writes first, regardless of their legitimacy, to check whether runtime definitions comply with statically computed reaching definition sets. Otherwise, corruption cannot be detected if the adversary leverages a write instruction not recorded in the runtime definitions table (RDT).

For this reason, we adopt a different approach to make targeted instrumentation possible. Instead of recording every write instruction, we check the value integrity of selected variables by allocating a shadow cell to each. Shadow cells are designed to store runtime values defined by legitimate instructions only. When a critical variable is legitimately defined, the instrumentation updates its corresponding shadow cell with the actual value written. When the same variable is used, the instrumentation checks whether the actual value read matches the shadow value recorded in the corresponding cell. If the values do not match, we can conclude that there is an attack because the variable should have been overwritten by an illegitimate instruction that is not statically computed. This approach substitutes instruction-based DFI checks with implicit *def-use* pair (reaching definition) checks bonded with variable values. Only one hypothetical scenario that would remain undetected (compared to the instruction-based approach) would be overwriting a variable value using the same existing value. However, the adversary cannot benefit from such a scenario, as he obviously needs a different value to perform the attack.

Thanks to this value-based approach that allows for targeted instrumentation, our proposal can thus avoid the overhead of checking non-critical variables. Besides, the value-based integrity replaces the cost of searching on instruction sets (i.e., reaching definitions) by a single value comparison. The only exception is pointers whose dereferences may define the same variable of interest. We suggest allocating separate shadow cells for data pointers to address such cases. In the event of a variable definition via legitimate pointer dereference, the instrumentation records the actual value written to the allocated pointer cell. Then, suppose there is a native use of a critical variable. In that case, the instrumentation checks both native variable and pointer cells for a matching shadow value. The search cost (i.e., the number of shadow cells checked) in such a scenario cannot be more than the number of pointers that may point to the given variable for the given program point. In contrast, searching for reaching definitions has to consider both pointer-based and native definitions that may reach from different control flow paths.

To explain how our value-based approach detects attacks and to discuss its equivalence to a naive DFI, we will often refer to Figure 5.5 in the following section. The first part (a) of this figure presents a fabricated code with different definitions and uses of imaginary program variables a, b. The code has three direct *definitions* (lines 1, 4 and 8) and two *uses* (lines 5 and 11) of a variable a. Also, it contains a potential indirect definition of variable a via pointer p (line 9), and a vulnerable instruction/function (line 2), which is supposed to define some other variable b, but used to corrupt variable a as an exploit of some memory bug.

The second part (b) depicts legitimate reaching definitions for each use of variable a (shown by black arrows) whereas a data attack scenario represents the corruption of variable a via defining instruction of variable b (an attack highlighted in red). The size of reaching definitions set also determines the search cost on each use (e.g., use of variable a at line 10 requires four compares as worst-case whereas the variable use at line 4 has only one).

In contrast, as illustrated in the third section (c) of the figure, our scheme indirectly checks reaching definitions using shadow cells. Instead of maintaining

an RDT, for a legitimate variable definition (lines 1, 4, 8), the scheme additionally creates a shadow value for the allocated cell of the variable (i.e., dotted outgoing arrows from definitions). In case of a variable use (lines 5, 11), our scheme checks whether the actual value loaded from the address matches the value previously stored on shadow cells (i.e., dotted incoming arrows to uses). If these values do not match, we conclude for a data-oriented attack since it means that an unexpected (non-reaching) instruction—not statically computed—should overwrite the actual variable address. For instance, if the adversary overwrites the variable a via exploitable `def b` instruction (line 2), because the shadow value will not be updated (due to the lack of instrumentation on `def b` or the instrumentation placed to update a different shadow cell), unmatched values will reveal the attack at the time of use of variable a (line 5 or line 10). Regarding value definitions via pointer dereferences, the value integrity scheme holds separate shadow cells for them. For example, `def *p` instruction (line 9) can legitimately define the value to be used by `use a` instruction at line 11. The instrumentation of defining instruction at line 9 creates a shadow copy for the allocated pointer cell, and the instrumentation of variable use at line 11 compares for both cells of variable a and pointer $*p$ regardless of the actual memory address holding value.

Since the proposed technique relies on shadow values to detect attacks, one concern may be the space requirements of composite variables such as arrays or strings. Although we have not identified those structures entirely as critical during our benchmark experiments, in such a case, we suggest using checksums to digest consecutive elements of composite variables similar to our proposal in Chapter 4.

5.4.2 Scope

Our approach recognises the overwriting of a trusted (critical variable) value by a non-critical variable instruction as an attack. Still, it deliberately ignores cases such that a non-critical variable instruction corrupts some other non-critical variable (e.g., modification of the user or environment input with the attacker payload). While this design choice forms a basis to the desired performance gain, it can

lead to the corruption of non-critical variables with values that do not satisfy path (semantic) constraints. However, the use of such occurrences for an attack is unlikely due to the following. First, for branch decisions dependent on the value or range checks of untrusted input, the control outcome would be already transferred to another variable as a trusted value, before the corruption point. Second, leveraging such state for the manipulation of control flow requires the program to have semantically redundant control structures (e.g., duplicate check within a nested control or loop structures). Otherwise, the program must have a direct use (e.g., `exec` argument) of the corruptible variable, which ideally should be hosting a legitimate user input, and a timely bug that can overwrite it between its check and its use—which would be a rare scenario.

Considering the lack of an established benchmark to evaluate effectiveness against data-oriented attacks, we evaluate the security promises of our approach based on the hardening of DOP attacks. In response to BOPC [32] automating DOP [31] attacks, we propose *Loop Protection Ratio (LPR)* as a metric of the reduced loop attack surface. Since powerful Turing-complete DOP attacks require the adversary to compromise at least a loop structure (i.e., gadget dispatcher) for arbitrary execution, the LPR metric, as a fraction of loop headers with instrumented condition variables, aims to assess the hardening of these attacks under our scheme.

5.5 Implementation

We have implemented a proof-of-concept of the design explained in Sections 5.3 and 5.4 to evaluate its performance promises primarily. Two LLVM passes[1] are implemented to analyse and transform the intermediate representation (IR) of C programs. Because LLVM IR is a language-agnostic representation, the analysis and transformation can also be adapted to the IR outputs of other unsafe languages that allow for managing its own application memory, therefore increases security problems. These passes selectively inject runtime checks. Hence, attacks targeting critical program variables can be detected.

[1] https://github.com/msgeden/truvin

```
1  void login(){
2     int authenticated=0;/*trusted value*/
3      int role_id;/*untrusted value*/
4      int login_attempt=0;/*trusted value*/
5      char pwd[STR_SIZE];/*untrusted values*/
6      char user[STR_SIZE];/*untrusted values*/
7      read(user,"Please enter username:");/*vulnerable*/
8      if (is_user_locked(user))
9          exit(ERROR_USER_LOCKED);
10     role_id=get_role_id(user); /*trusted value*/
11     while (authenticated==0 && login_attempt<=MAX){
12         read(pwd,"Please enter password:");/*vulnerable*/
13         if (check_credentials(user,pwd))
14             authenticated=1;/*trusted value*/
15         login_attempt++;/*trusted value*/
16     }
17     if (authenticated==0 && login_attempt>MAX){
18         lock_user(user);
19         exit(ERROR_USER_LOCKED);
20     }
21     if (authenticated){
22         if (role_id<=SYSTEM_ADMIN)
23             generate_privileged_session(user);
24         else
25             generate_unprivileged_session(user);
26     }
27     return;
28 }
```

Figure 5.6: Vulnerable program code that forms the basis for different data attack scenarios.

5.5.1 A Concrete Example

To illustrate how those passes work, we use a vulnerable C program that represents a typical login system, as seen in Figure 5.6. This program contains five variables out of which two pwd and user arrays host the login credentials. Other authenticated, role_id and login_attempt variables are used by control and loop structures. The program loads untrusted user credentials to the memory through a vulnerable read function. The attacker can exploit this function to bypass the credential check (i.e., authenticated) or he can reset login_attempt to perform brute-force attacks for password discovery. Also, he can modify role_id for privilege escalation.

```
line 4: int login_attempt=0
store i32 0, i32* %4, align 4
.....................................
line 10: role_id=get_role_id(user)
%14 = call i32 @get_role_id(i8* %13)
call void @vi_def_32(i32 %14, i16 2)
store i32 %14, i32* %3, align 4
.....................................
line 11: while( ..login_attempt<=MAX)
%19 = load i32, i32* %4, align 4
call void @vi_use_32(i32 %19, i16 3)
%20 = icmp sle i32 %19, 5
.....................................
line 15: login_attempt++
%31 = load i32, i32* %4, align 4
call void @vi_use_32(i32 %31, i16 3)
%32 = add nsw i32 %31, 1
call void @vi_def_32(i32 %32, i16 3)
store i32 %32, i32* %4, align 4
.....................................
line 17: if(..login_attempt>MAX)
%37 = load i32, i32* %4, align 4
call void @vi_use_32(i32 %37, i16 3)
%38 = icmp sgt i32 %37, 5
br i1 %38, label %39, label %41
.....................................
line 22: if (role_id<=SYSTEM_ADMIN)
%45 = load i32, i32* %3, align 4
call void @vi_use_32(i32 %45, i16 2)
%46 = icmp slt i32 %45, 2
br i1 %46, label %47, label %49
```

Figure 5.7: IR instrumentation on the slices of role_id and login_attempt variables.

5.5.2 LLVM Passes

LLVM compiler enables developers to analyse, optimise and transform their programs. As the core strength, it uses a language- and target-independent intermediate representation (IR). LLVM IR is a high-level strongly-typed assembly language with RISC-like instructions, many of which are in three-address code. It uses partial static single assignment (SSA) with an infinite virtual register set and assumes two kinds of variables which are *top-level* and *address-taken* variables.

5.5.2.1 Trust Propagation

Our first pass analyses how the trust propagates throughout the program, as explained in Section 5.3. Although it allows the analyst to define additional trust sources (e.g., database reads), we have used programmer-defined values as the main trust source during our benchmark experiments. This pass propagates the trust through all top-level and address-taken IR variables. However, only address-taken variable operations are instrumented. Fig. 5.7 illustrates the trust propagation on the IR slice of role_id and login_attempt variables in Fig. 5.6. Operands highlighted in different colours represent pre- and post-instruction states of trust. Differently from other variables, role_id starts hosting a trusted value upon return of get_role_id(user) function. Since this function returns only programmer-defined constants (due to the limited number of roles), it is discovered as a trust source by the analysis pass as it should. This pass provides the information of trusted values available on the entry and the exit of each IR instruction within a flow-sensitive setting. It thus determines which *def-use* pairs (*store-load*) operate with trusted values and must be instrumented by the following transform pass.

5.5.2.2 Value Integrity Checks

Our second pass transforms the program IR by injecting function calls to check the value integrity of identified variables. As explained in Section 5.4.1, these calls either store shadow values for variable definitions or load existing shadow values to check whether they match with the actual ones during variable uses. As an example, Figure 5.7 demonstrates injections placed as shadow operations of role_id and login_attempt. These functions wrap the necessary instrumentation, which is inlined during compilation. Depending on the type of access, injected calls are placed before the *store* instruction (vi_def_xx) or after the *load* instruction (vi_use_xx). Both function groups accept two kinds of parameters. The first argument holds the exact copy of the variables (e.g., value operand of *store* and the result operand of *load*). The second one is the variable identifier(s) that help to locate the corresponding shadow cell(s).

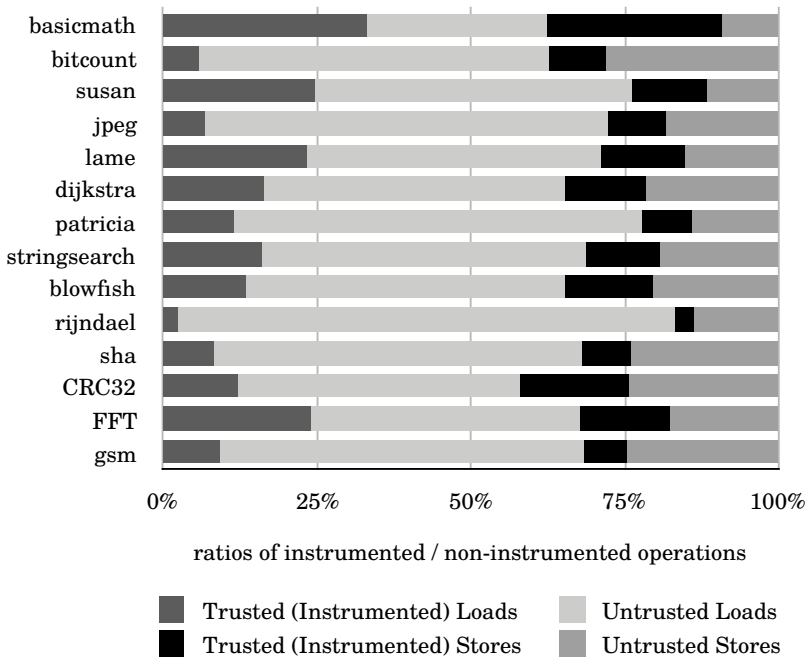


Figure 5.8: Ratio of instrumented memory instructions.

5.6 Evaluation

This section first evaluates the performance gains regarding runtime and space overheads. Next, it analyses the security guarantees provided.

5.6.1 Performance

For performance evaluation, we have experimented with MiBench [158], which is the most popular open source uni-processor benchmark suite. To contrast the performance of targeted instrumentation with a naive approach, we have compared our approach against fully instrumented benchmark programs. The full instrumentation places injections for all memory operations. It is considered as a performance approximation of the naive DFI scheme, while not making any security promises.

Regarding the instrumentation avoided, Figure 5.8 presents the ratios of memory instructions that operate with trusted and untrusted values. Although the metrics vary depending on the program and the operation, only 22% of the memory operations require instrumentation for the entire benchmark. This corresponds to

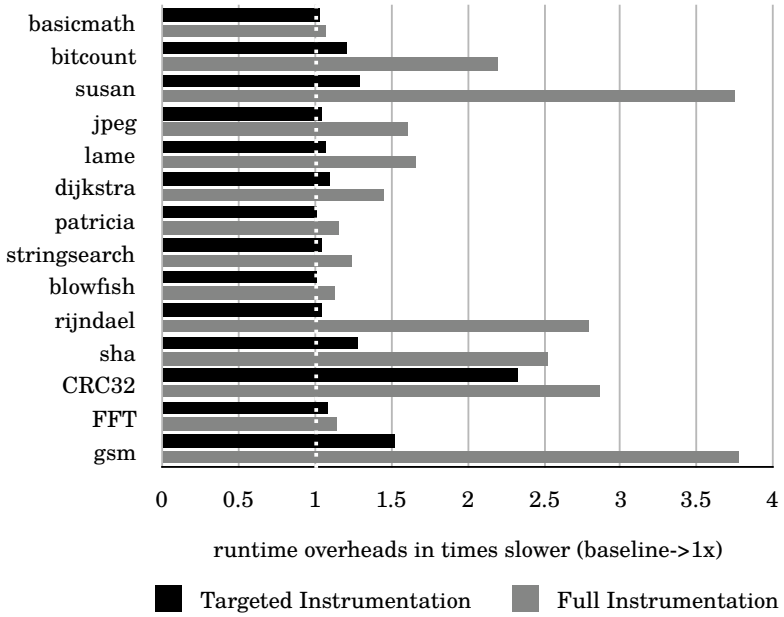


Figure 5.9: Runtime overheads of benchmark programs with targeted and full instrumentation.

12849 out of 56694 (*load:* 7508/40864 and *store:* 5950/15830), which implies that the great majority of operations can be left non-instrumented.

Although performance gain arises from operations left non-instrumented, the number of executions of instrumented ones determine the actual overheads. For the benchmark suite, our scheme has produced only *28%* runtime overhead. In contrast, the full instrumentation has incurred 121% overhead, which is a close approximation of the naive DFI scheme [4] reporting 104% overhead for a different set of benchmark programs. Figure 5.9 presents detailed runtime overheads of each transformed benchmark program to compare our scheme against fully instrumented programs. Performance gains vary from 1.4x to 44x depending on the benchmark program. Regardless of variance, the selective instrumentation provides 4.3x performance gain for the whole benchmark suite. Regarding the space overheads, shadow memory that hosts only critical variables, requires 8x less memory than the allocation made for full instrumentation. Moreover, selectively instrumented binary sizes with inlined calls are only 40%, whereas fully instrumented binaries are 134% greater than non-instrumented binaries.

Program	Trusted	Untrusted	LPR
basicmath	17	1	94.4%
bitcount	6	0	100%
susan	47	0	100%
jpeg	398	61	86.7%
lame	348	18	95.1%
dijkstra	7	0	100%
patricia	9	1	90%
stringsearch	19	2	90.4%
blowfish	36	0	100%
rijndael	10	0	100%
sha	10	0	100%
CRC32	3	0	100%
FFT	12	0	100%
gsm	62	2	96.8%
TOTAL	984	85	92.1%

Table 5.1: Loop headers with instrumented (trusted) and non-instrumented condition variable operations. *LPR*: Reduced loop attack surface for DOP attack elimination.

5.6.2 Security Analysis

For a successful attack, the adversary must overwrite a trusted value of critical variable by exploiting an illegitimate instruction that belongs to either a non-critical variable or another critical variable. The first scenario, an illegitimate data flow from untrusted agents to the critical variable addresses (trusted domain), will be caught due to an outdated shadow cell of the variable of interest. The second scenario, an illegitimate data flow among the critical addresses, will also be revealed due to unmatched shadow values, since the instrumentation of the exploited instruction would update only its corresponding shadow cell.

5.6.2.1 DOP Attacks

Table 5.1 demonstrates the number of loop headers (i.e., program statement that decides on the loop iteration) whose condition variables are identified as trusted (critical). The ratio of variables with trusted values to all variables corresponds to the LPR metric explained in Section 5.4.2. Because variable instructions operating with trusted values are instrumented, DOP attacks in need of a gadget dispatcher utilising these loops (variables) will be caught. For the entire suite, our scheme

promises to reduce the loop attack surface by 92.1% as it defends all the loop headers for 8 out of 14 benchmark programs (100% LPR). The remaining loop variables with untrusted values could genuinely be part of or dependent on user inputs. Or some of them might be defined via external functions that need to be stated as the trust source by the analyst. However, in either case, it provides strong evidence that our scheme successfully identifies variables critical to the program flow and catches data attacks without any programmer intervention.

5.6.2.2 Real-world Scenarios

Although there is no established benchmark to measure the effectiveness of our scheme, we discuss its potential based on some real-world exploits (see Appendix A.1 for their code snippets) introduced in related papers [31, 151, 159, 160]. The first is a integer overflow vulnerability found in many *SSH* implementations [161], which enables overwriting the *authenticated* flag to bypass the authentication process. Similarly, a *Chrome* bug [162] in the renderer process can be exploited to overwrite the critical *m_universalAccess* flag of security monitors to bypass the SOP (same-origin policy) enforcement. Our scheme can detect both attacks as they overwrite decision-making variables defined by the programmer.

Two examples *wu-ftpd* [163] and *sudo* [164] suffer from format string vulnerabilities. These vulnerabilities allow the attacker to modify variables hosting user ID of calling processes, whose corruption would result in privilege escalation. Both cases would be caught as long as *getuid()* system call is defined as an external trust source.

Furthermore, *Null httpd* server bugs [165] can corrupt the path string of CGI-BIN directory, which makes remote executions possible. Another heap corruption vulnerability in *telnet* daemons [166] can be exploited to modify the configuration string *loginprg* that states the path of an executable responsible for the user authentication. An extended design that can digest string array similar to what we have proposed in Chapter 4 would be able to detect those scenarios.

As a rare scenario that would stay undetected, *ghttpd* [167] web server holds a stack buffer overflow bug in between the check and the use of a URL string. An

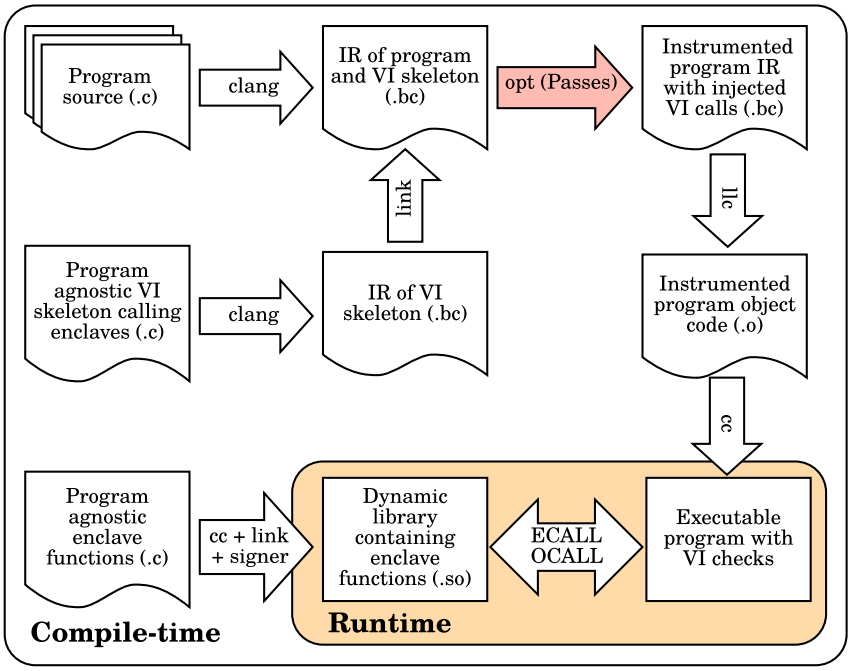


Figure 5.10: Compile time and runtime stages for SGX adaptation.

attacker that overwrite the URL string would make its prior sanity check useless. This would result in remote shell execution. This is the only exceptional case that would be missed as discussed in Section 5.4.2. But our scheme can still eliminate Turing-complete capabilities (i.e., LPR), and most of the real-world (6 out of 7) exploits as non-arbitrary attack examples.

5.7 Intel SGX Adaptation

An adversary can also try to disclose and modify the shadow values generated by the instrumentation process itself. For such scenarios, a hardware-based trusted execution environment (TEE) such as Intel SGX can provide strong isolation against even bugs exploitable from higher privileges (e.g., kernel).

Intel SGX provides special memory regions, called *enclaves*, for the isolation of selected program code and data. A program with SGX features consists of two parts that are application (untrusted) and enclave (trusted) components. Switches between the two are performed by special interfaces such as *ECalls* and *OCalls*. Enclaves can mitigate control-oriented attacks on the code running in those regions

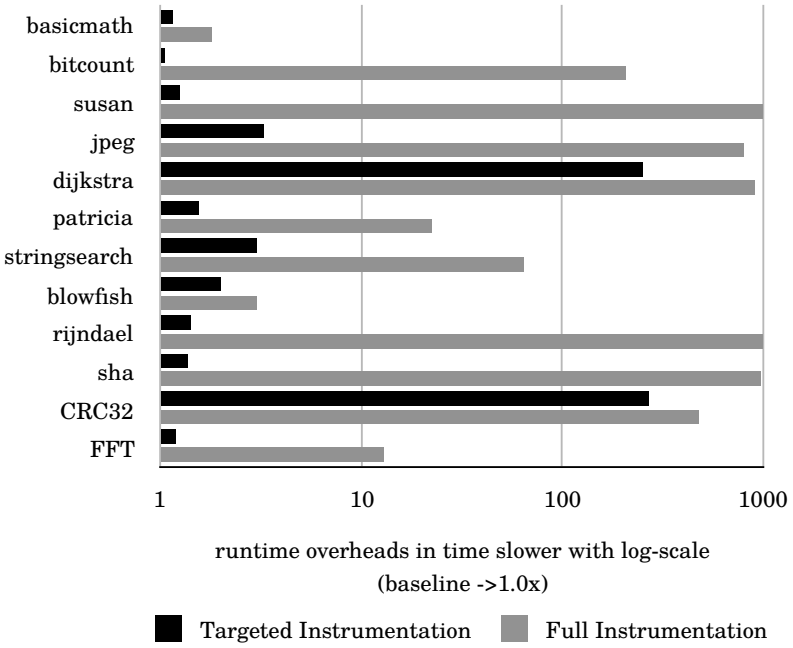


Figure 5.11: Runtime overheads of enclave-hosted shadow data (-O1 with x64).

by encrypting the code and code pointers. However, they cannot eliminate a data-oriented attack performed as a relative address corruption since the layout of variables within the enclave would not be different.

5.7.1 Program-agnostic Enclaves

Normally, the programmer decides on which part of the code and the data should be in the enclave. Our adaptation (see Figure 5.10) enables the use of enclaves for any program in an automated way without the programmer intervention. To achieve this, we have first implemented an instrumentation skeleton containing the functions wrapping enclave calls (ECalls). This part is statically linked to the program IR, which is later used by the passes. In the final stage, the transformed program IR is compiled into object code and translated into an executable program. On the enclave side, program-agnostic functions are compiled and signed as usual, the final output of which is a dynamic library loaded by the instrumented program executable.

5.7.2 Switch Overheads

Enclaves provide strong security guarantees. However, this level of security does not come free and brings huge performance penalties due to the switches between enclaves and outside memory. Hence, a scheme simply using enclaves to allocate shadow memory for all program variables would not be very convenient due to costs triggered by every memory access. Our targeted approach aims to make TEE-based isolation possible for a limited number of decision-making variables left after additional register allocations.

Even though unoptimised binaries (-O0) using enclaves have demonstrated that targeted instrumentation of critical variables can provide performance gains up to 29x, programs that heavily iterate loops via address-taken counter variables increase the switch overheads unnecessarily. To address this issue, we have explored the use of register allocations as means for cutting those switch overheads, by substituting possible address-taken loop iterators with registers. We highlight that those register-hosted variables do not downgrade security guarantees as long as their values are not spilled to the memory. Our experiments have shown that these allocations can significantly reduce switch costs for a more realistic compilation setting (-O1). Compared to the scenario where all memory operations are instrumented, relative performance gains vary from 1.6x to 5556x (see Figure 5.11) depending on the program, with a total 5.3x reduction rate for the whole suite (excluding *lame* and *gsm* programs due to the library conflicts of SGX and the program).

5.8 Summary

This chapter presented a novel runtime protection mechanism to address data-oriented attacks exploiting memory bugs in C/C++ programs, without having to instrument every memory operation. Our proposal focuses on program variables that are only critical to program semantic, and avoids redundant checks on preliminary or intermediary targets not enough to perform an attack. This means that we only

need to instrument a fraction of the variables, thereby saving both CPU time and memory, while retaining similar security guarantees.

To make this idea work, this chapter introduced a novel method to identify critical program variables based on the trustworthiness of their value agents, i.e., the programmer as a trusted or the user as an untrusted agent. In addition to the variables whose values are directly defined by trusted agents, we have proposed a static flow-sensitive trust propagation analysis to discover other emerging variables still dependent on those (as depicted in Figure 5.3). Based on the information extracted, the scheme transforms a (potentially) vulnerable program by selectively injecting runtime checks on the instructions that must operate with trusted values. This ensures that if a trusted value of a critical variable is corrupted, our scheme will detect it, but without the overhead of checking every variable address in the program. Benchmark experiments show that such a targeted approach can provide a performance gain of around 4.3x.

Similar to many runtime protections, the instrumentation data must be well-protected within the same address space for this scheme to fulfil its security promises. However, without hardware- or TEE-based isolation, we have to count on weaker hiding (randomisation) [3] or performance-costly SFI [11] methods. On the other hand, hardware-based TEEs such as Intel SGX or ARM TrustZone can provide special regions protected even from OS bugs. But using those primitives is not very promising for many programs due to their switch costs triggered by each shadow operation, as discussed in Section 5.7. This issue has led us to consider using other hardware but faster components (i.e., CPU registers) as trusted storage for protection of integrity-critical data. The following chapter takes this idea forward and proposes a practical runtime protection scheme that relies on the idea of keeping critical variables safe in CPU registers.

"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."

— Edsger Wybe Dijkstra

6

Leveraging CPU Registers for Protection of Runtime Data on the Stack

CPU registers are not addressable in the same way that memory is, which also makes them immune to memory attacks and manipulation by other means. In this chapter, we take advantage of this to protect critical program variables from corruption, without asking for a memory isolation primitive. The scheme presented in this chapter can effectively address control- and data-oriented attacks targeting the stack, even by adversaries with the full knowledge of program memory. While the primary design focus in this chapter is security, performance is also important for a runtime protection to be adopted in practice. The solution given in this chapter still benefits from the performance gain that is normally associated with register allocations, and it provides a practical protection.

6.1 Introduction

Memory bugs continue to be one of the root causes of software security problems, especially in applications developed using unsafe languages, such as C and C++, which are commonly used in systems programming and performance-critical applications. Since there is no built-in memory safety in those languages to prevent unintended access to critical program data, an attacker exploiting a memory bug

in the program (e.g. stack buffer overflow) can overwrite control and data objects beyond the abstraction given in the source code.

Several schemes have been proposed to mitigate the consequences of these memory bugs. The majority of these focus on *control-oriented* attacks in which code pointers are targeted. For example, stack canaries [168] place random values on the stack to detect overflows onto return addresses. But these canaries fail to catch well-targeted corruptions, e.g., format string attacks, that can access certain addresses and leave the canary untouched. More powerful control-flow protections, which do not make assumptions about how memory corruption happens, exist such as CFI [2] and CPI/CPS [3]. Those either use a shadow stack to detect corruptions of (shadowed) control data or a safe stack to protect them from being altered. These control-flow protections do not cover *data-oriented* attacks that selectively target non-control data, for example, a function argument or a condition variable deciding on the execution of a privileged branch. Proposed defences against those attacks, e.g., data-flow integrity (DFI) [121], generally require a more thorough check of all stack accesses in addition to code pointers, and in the process they introduce high performance overheads.

Regardless of their limitations, current proposals for control- and data-oriented attacks face three common challenges in general. The first is the performance overhead due to the instrumentation required for memory operations, which worsens as the coverage expands (i.e., non-control data) as we have tried to address in Chapter 5. The second challenge is that their success is dependent on how well the instrumentation data (e.g., shadow stack) or segregated data (e.g., safe stack) is protected within the same program space. Current techniques hide the location of those through randomisation or implement some access policies for them. However, integrated attacks that reveal or search the location of instrumentation data can break the schemes' promises [10, 34]. The final third issue is the lack of deployability by different device types and architectures. For strong assurance, many proposals require modifications to ISA [9, 104, 122, 169] or require features provided by a specific architecture (e.g., Intel MPX [170]), which makes them

deployable only for future devices or for a small portion of existing systems. Also, the majority of defences are designed for high-end devices with a reliable operating system, whereas primitive architectures and embedded systems (e.g., bare-metal) are generally ignored.

This chapter presents RegGuard, a new runtime protection scheme to harden C/C++ programs against stack-based corruptions. It leverages CPU registers to protect critical variables in use, with further integrity assurance even if their states are saved to the stack. Our scheme successfully addresses all three challenges mentioned above and differs from previous proposals by providing practical and robust protection against both control- and data-oriented attacks. It is practical because our scheme is designed as a software-only solution that does not require any new hardware. Besides, replacing memory accesses with registers still compensates for most of the overhead. It is robust because CPU registers, as unaddressable storage units, provide a strong hardware root of trust for the storage of critical data in use. Thanks to our cryptographic assurance that covers register data at rest on the stack, we do not need to worry about integrated attack scenarios, as it does not generate any instrumentation data that must be hidden or protected in program memory. Lastly, because our scheme is built on one of the fundamental building blocks of computers (i.e., CPU registers), it can be adapted to different device types and CPU architectures, including both modern and legacy systems, with trivial changes to the software stack.

To verify that the integrity checks introduced by RegGuard do not make the performance of the resulting binary unusable, we have implemented a proof-of-concept using LLVM compiler for the ARM64. ARM has been one of the most dominant architectures of mobile phones and microcomputers for a while, which makes it a good platform for testing performance. Our experiments have shown that register allocations can improve both the security and performance together with a surplus within the range of 13% to 23% on average compared to purely performance-based optimisations, whereas almost all have had better performance than non-optimised versions.

The rest of the chapter is organised as follows: Section 6.2 explains our motivation, system and threat models. Section 6.3 presents the design of our scheme. Section 6.4 provides the details of proof-of-concept implementation, whereas Section 6.5 provides an evaluation based on this implementation. Lastly, Section 6.6 discusses certain design decisions and further extensions that can be taken forward.

6.2 Problem Setting

Separation of memory into (read-only) code and (non-executable) data addresses through W⊕X in most systems has made it harder to perform simple code-corruption and -injection attacks. In response, more sophisticated code-reuse scenarios such as ROP have become more prevalent. Although control-flow protections [2, 3] mitigate attacks on control data (i.e. code pointers), they fail to capture more challenging cases where non-control data objects (e.g. condition variables) are targeted. Addressing those attacks has proven difficult in practice as they either introduce heavy instrumentation costs [4] provisioning each memory (data) access or require expensive hardware changes [9]. Furthermore, software-based approaches must secure their instrumentation data within the same address space. However, commonly used techniques such as hiding can be circumvented when the location of the data is revealed through an integrated attack [10]. This chapter takes those drawbacks into account while mitigating both attack classes.

6.2.1 Motivation

In order to modify a stack object, the attacker must either overflow some buffer onto the target object (i.e. relative address attack) or take over a data pointer first to overwrite it (i.e. absolute address attack). CPU registers are immune from such attacks since they cannot be addressed in the same way we address memory.

However, to use CPU registers as a protection mechanism, we have to solve a couple of challenges. First, we must use them for security while still allowing them to serve their primary purpose, namely as a fast storage mechanism for data in use to reduce execution time. Second, we have to find a way to leverage the

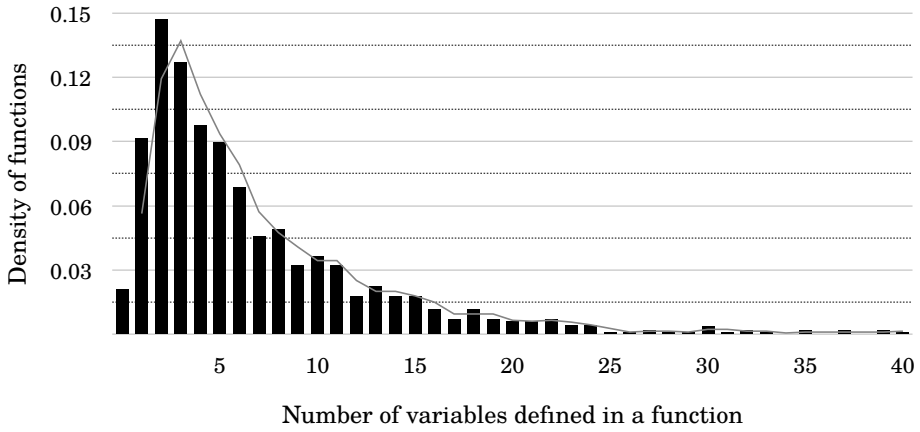


Figure 6.1: Probability distributions of variable counts per function.

limited capacity of the registers to protect all the relevant variables. Simply using CPU registers as program-wide (interprocedural) storage would put a hard limit on the number of variables allocated, whereas register states that are saved to the stack during function calls void their immunity against potential corruptions. Therefore, we need a global (function-level) allocation scheme that can employ the same registers for each call without undermining security. With such integrity assurance, CPU registers can provide enough storage to secure critical control and data objects on the entire stack.

To provide insight into the coverage that such a protection scheme can provide, Figure 6.1 shows the number of variables found per function in a large representative set of benchmark programs provided us with more than a thousand functions in total. We have used the same set of programs for our performance evaluation in Section 6.5. We remind the reader that as the program size gets bigger, its functions should tend to get smaller due to the challenges of managing complexity by the programmer. As seen, 93% of the functions have less than 16 variables, and 99% have less than 32 variables. Considering the average number of variables (6.9) and arguments (2.6) found per function, most modern CPUs provide enough registers (with 16/32 GPRs and 32 FPRs) to secure those objects as potential attack targets. Note also that these counts represent all reference and primitive variables found in a function at any point, and do not take the live ranges into account, so the number of concurrently

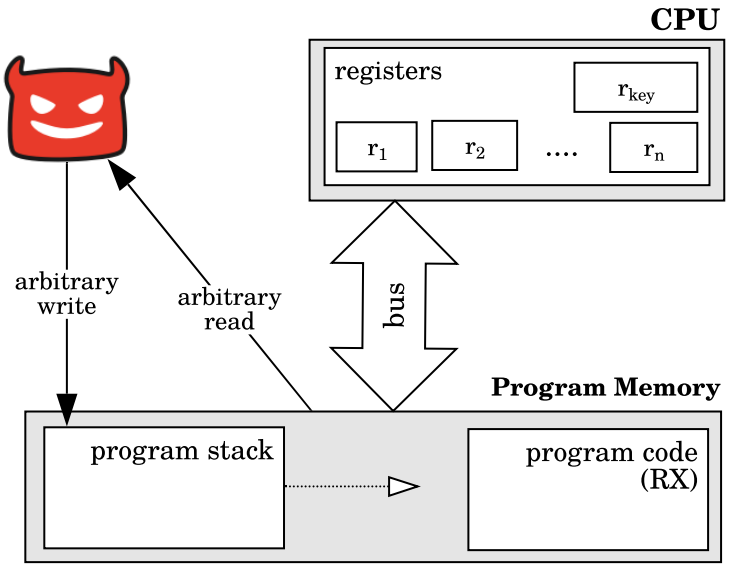


Figure 6.2: Overview of the system components and adversary model.

live ones would be smaller. In Section 6.3.1, we show how it is still possible to deal with the rare event that this number exceeds the number of available registers.

6.2.2 System and Adversary Model

In our model, the CPU is trusted and provides limited but secure register storage. Regarding program memory, the system (see Figure 6.2) ensures code integrity through non-writable (RX) addresses, which can be provided by a flash memory or page-level (e.g., W⊕X) protections, depending on the device setting. The CPU has n registers available (r_1-r_n) for the scheme. The system dedicates a specific register (r_{key}) to store the key, for instance, a single FPR that is never saved in the program memory. We deliberately avoid making assumptions about the type of device and its architecture. It can be a single-threaded bare-metal environment as well as a multi-threaded one with an operating system, the kernel space of which is trusted by the user processes. As long as the system has the necessary CPU registers and ensures the integrity of program code, our scheme is applicable to different architecture and software/firmware instances. We assume that the software stack running on the device can be recompiled and modified as required, without asking for any change in hardware.

The adversary's goal is to manipulate the program runtime by modifying critical control and data objects on the stack, although program termination does not constitute a successful attack. For instance, he can target a code pointer such as a return address or a function pointer to hijack the program's control flow. Alternatively, he can overwrite a non-control data object, for example, a condition variable to manipulate the control flow indirectly. We assume a powerful adversary that has full read access to any part of memory at all times (including the stack), as well as write access to any address on the program stack. We are not going to explore how such read and write capabilities can be achieved in practice; we just grant the adversary this power. We do assume that the adversary cannot intervene in the compilation process and cannot modify program code in the non-writable code segment, which includes our instrumentation as part of it.

This model extensively captures control- and data-oriented attacks. It addresses both code-reuse attacks bypassing DEP, and more challenging data-oriented attacks that can otherwise circumvent control-flow protections (e.g., CFI). This model also covers a wide range of scenarios on how the adversary can interact with the program memory. In contrast to protections relying on random values (e.g., stack canary [168]) or random addresses (e.g., safe stack [3], ASLR [25]), this model covers integrated attacks [10] that exploit memory disclosure bugs first.

6.3 Design

During the compilation process from source code down to machine code, the compiler has to map variable objects to either memory addresses or CPU registers. Since registers are safe from memory corruption and can be accessed very fast, we would prefer to put all variables in registers. However, this is not always possible as there can be more (simultaneously live) variables than available registers (i.e., high register pressure), especially for CPU architectures suffering from register scarcity. Therefore, we must first ensure that the compiler prioritises a variable that is more likely to be targeted by the attacker for register allocation. Second, even if all variables of a function are assigned to registers, their values will be saved to the

program stack during a function call, to make the registers available to the new function. Because these saved values can be overwritten on the stack, we must do something to guarantee their integrity.

6.3.1 Security-Oriented Allocations

Similar to the conventional spill cost that estimates the performance burden of a variable left in memory, we assign a *security score* to each variable to ensure that a register is primarily allocated to a variable that is more likely to be attacked. In contrast, a security score is a compile-time estimate of how critical a function variable is to the runtime integrity. Variables with higher security scores are thus prioritised for register allocation and are included in the integrity checks designated for register values saved in the stack during a function call, as explained in detail in Section 6.3.2.

6.3.1.1 Security Scores

Our scheme considers the variables listed below as primary attack targets that must be prioritised during register allocations. It assigns a security score to each according to the given order (i.e., the first in the list has a higher score).

1. code pointers, e.g., function pointers,
2. data pointers, i.e., variable addresses,
3. programmer-defined variables, e.g., `is_admin=1`,
4. condition variables, e.g., `if(authenticated)`.

Pointers have the highest scores as they are the most common attack vector for powerful attacks. If not caught, the corruption of a code pointer such as an indirect branch or a call target can result in arbitrary execution, while a data pointer can be used to access or modify other data objects on the memory (i.e., absolute-address attack). Next comes the variables whose values are set from the code and condition variables used for branch decisions. We remind the reader that programmer-defined variables are different from the constants evaluated and eliminated at compile time

```
1  ...high register pressure...
2  int (*func_ptr)(const char *,...) = &printf; /*function pointer*/
3  int is_valid=0; /*decides on control flow*/
4  int drop_stats=0; /*no critical use*/;
5  int max_trial=read(); /*user defined data*/
6  char data[SIZE]; /*buffer hosting untrusted environment data*/
7  /*the user has a legitimate control over the loop iterations*/
8  while (--max_trial>=0){
9      /*vulnerable function*/
10     read_buffer(data);
11     if (check(data){
12         is_valid=1;
13         break;
14     }
15     drop_stats++;
16 }
17 if (is_valid==1) /*decides on control flow*/
18     do_process(data); /*critical task*/
19 (*func_ptr)("trials of %s is %d", data,drop_stats); /*print stats*/
20 ...high register pressure...
```

Figure 6.3: Code under register pressure for the given scope. For security, registers are allocated to func_ptr and is_valid first instead of less critical max_trial and drop_stats.

by the optimisations. A programmer-defined variable whose all possible values are hard-coded actually represents the programmer's intention. Although those are generally used as condition variables, they allocated first compared to ones defined from unknown origins. This is because an attacker would not benefit from corrupting a data object that is already controlled or defined by the user or environment as we have discussed in Chapter 5. Return addresses, return values, and function arguments are also assigned to registers. But they are excluded from this scoring and selection process because the calling convention in place already dedicates registers for them.

Figure 6.3 exemplifies how our *security scores* differ from conventional spills costs. This code depicts a high register pressure for the given scope. Normally, a conventional scheme would allocate registers to drop_stats or max_trial variables first for better execution times as they will be accessed by each loop iteration. However, from the security point of view, RegGuard considers that func_ptr and is_valid should be given registers primarily. Alteration of func_ptr as a code

```
Function securityScore(var)
    var.score ← 1;
    if var.type is a pointer type then
        var.score ← var.score + 4;
        if var.uselist has a branch instance then
            var.score ← var.score + 1;
        end
    else if var.type is an integral type then
        if var.deflist has an immediate assignment then
            var.score ← var.score + 2;
        end
        if var.uselist has a comparison instance then
            var.score ← var.score + 1;
        end
```

Figure 6.4: Pseudocode of security score calculations.

pointer can result in illegitimate execution of sensitive system functions, whereas modifying is_valid flag, which is both a programmer-defined and a condition variable, would manipulate branch decisions as a data-oriented attack. On the other hand, max_trial which is defined externally (e.g., the user) or drop_stats that does not affect control-flow of the function are not identified as critical.

In contrast to spill costs given based on the use densities of variables, security scores that represent the likelihood of a register candidate to be attacked are designed as a fast intraprocedural static approximation considering the type of a variable, its value agents and use purposes. Therefore, a security score must be uniformly associated with different live ranges of a variable. In other words, the scores should not be localised for different ranges. Algorithm 6.4 shows how those security scores are calculated to rank register candidates in an order that would maximise security by taking those properties into account.

6.3.1.2 Allocation Process

As a global register allocation scheme, our scheme works at the function level to reuse the same register file repeatedly for each call and accommodate more critical data objects in the registers. The allocation technique to be used (see Chapter 2 for different options) should be chosen according to the features of the compiler.

For instance, the compilers using single static assignment (SSA) form, such as LLVM, generally implement custom linear techniques with faster compilation times, whereas other compilers can provide graph-colouring as the default option. We highlight that the choice of allocation method, which some compilers provide as a configurable option, is a separate issue from the problem our scheme addresses. And it does not have any impact on the applicability of our scheme as long as conventional spill costs are replaced by the security scores proposed. Any global allocation technique provided by the compiler can thus be preferred.

We recall that registers are actually allocated to the live ranges of variables. A *live range* describes the instruction or basic blocks scope ranging from a value definition to its all uses for the same definition. Live range definitions allow us to reuse a register for different variables whose ranges do not interfere with each other. A variable can have multiple live ranges with potential gaps in between, where each starts with a new definition. The variable does not have to occupy a register during these gaps. Therefore, allocation schemes generally use them for more optimal allocations. Such cases also benefit our scheme without undermining its security promises, since the attacker cannot benefit from overwriting a variable value that will be later redefined before its use. The attack surface thus gets smaller as the registers are utilised better. This can be meaningful for architectures suffering from register scarcity.

Figure 6.5 depicts how RegGuard should allocate available registers to the variables using security scores; so decides which variables to be protected. This example considers a scope under high register pressure with two available registers reg_1 and reg_2, and three variables, the live ranges of which interfere as shown. The security scores are represented by colour tones, var_3 is the most critical target, followed by var_2, whereas var_1 has the lowest score. Using security scores, the scheme priorities two registers to var_3 and var_2 and spills var_1 when required. However, the allocation method can still utilise gaps (i.e., instructions that var_3 and var_2 do not interfere), where a register become temporarily available for var_1. Those splits not only enhance the performance but also provide a better reduction

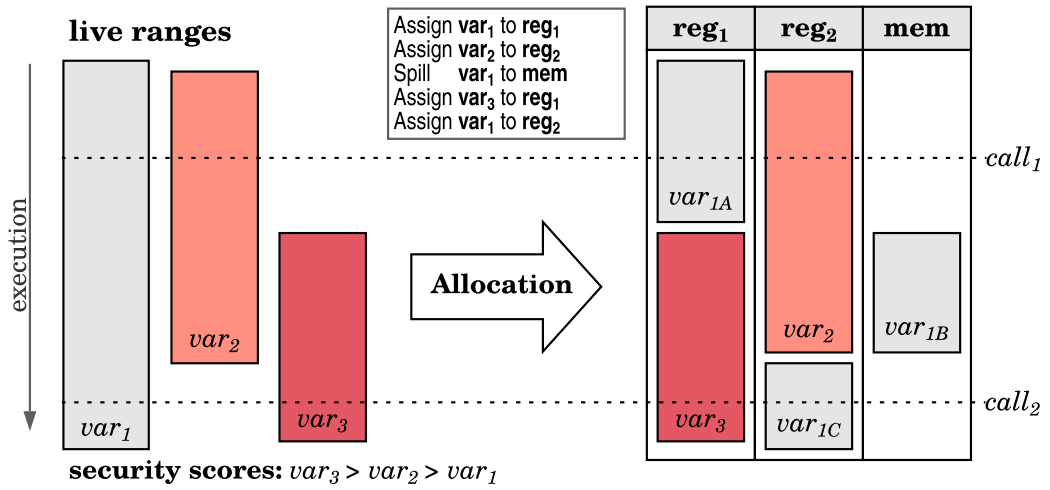


Figure 6.5: Security-oriented register allocations under register pressure.

of the attack surface. For instance, regardless of its criticality, var_1 in the example can thus enjoy both performance and security promises, even if for a short time. And it will be safe during the execution of function calls, $call_1$, $call_2$ depicted as potential attack vectors. Suppose such a case occurs while a critical variable range is left in the memory. In that case, the compiler displays a warning message to guide the programmer to review the code.

6.3.2 Integrity of Saved Register Values

The program can save a register value to the stack for one of two reasons. The first is to free up a register for a more critical variable within the same function. These register spills can happen only under high register pressure, and the decision of evicting a register in use for another variable is guided by the security scores described in Section 6.3.1. The second more common reason, which should take care of, is a new function call that triggers the eviction of registers for the callee function. Those register states that belong to the caller's execution are saved to the stack either by the caller at call sites or by the callee as part of its prologue code. The decision of which registers must be saved/restored by the caller and the callee is mainly described by the calling convention. Regardless of the calling convention in use, any register state saved to the stack becomes vulnerable to memory corruption.

Table 6.1: Variance of register saves during the callee function.

Target Type	Variance
Variables (Not Addressed)	Static
Variables (Locally Addressed)	Static
Variables (Called by Reference)	Dynamic
Temporaries	Static
Arguments	Static
Return Addresses	Static
Frame Pointers	Static
Return Values	Static

Therefore, RegGuard implements integrity checks on those to ensure that they are restored back to the registers without any corruption on return.

6.3.2.1 Invariance of Saved States

The integrity assurance covers saved register states that must not change during the execution of a callee function. Table 6.1 presents an overview of those as potential targets. The only exception is the values that can be legitimately modified by the callee, usually an updateable value passed as a call-by-reference argument. Otherwise, RegGuard protects local variables, return addresses, frame pointers, temporaries, function arguments, and return values that can be leveraged for an attack. With a fine-grained (e.g., flow-sensitive) pointer analysis [171, 172] that distinguishes pointers with only local accesses from call-by-reference arguments, where the latter must be destroyed following the call instruction, RegGuard can also ensure the integrity of locally addressed variables whose values must not change during the callee's execution. Because pointer analysis is a separate research problem that is orthogonal to the focus of this chapter, we will not discuss this issue further.

6.3.2.2 Integrity Checks

We recall that the data in use on registers are already safe from attacks, and the only attack surface left is register values saved to the stack. Therefore, RegGuard employs a cryptographic keyed hash (MAC) to guarantee that those saved register values have not been modified while at rest on the stack. Prior to the execution of

a function body, our scheme computes a reference tag from register objects being saved to the stack. This tag value is also kept in a specific register unless the callee function makes another call. Upon completion of the function body, a new tag is generated from the actual objects that are being restored to the registers. This tag is compared against the reference tag previously generated from saved objects, any corruptions on those can thus be revealed. For a function call consisting of both caller- and callee-saved registers, this is a two-step process connected. The first tag generation/verification of caller-saved registers is managed at the tails of call sites, whereas the following tag digesting callee-saved registers is created/checked by the function prologues/epilogues.

Function-wise, RegGuard digests each call frame using a single tag value. Program-wide, because we save the tag register to the stack with other registers and include it in the next tag calculation, our scheme actually creates a chain of tags that provides (almost) a complete stack image on a single register. Although we still rely on the key for integrity checks, this chain prevents the attacker from replaying a (standalone) call frame and its corresponding tag for a different call context. Thanks to the control over the compilation process of the software stack, we remind the reader that the key register is never saved to the same program/process memory, which is adequate to authenticate any tag restored from the memory that serves as the integrity proof of restored objects. With a single key kept secret on a dedicated register and MAC calculations that are part of non-writable program code, RegGuard enables the use of register file as an integrity-guaranteed storage for each function call.

Figure 6.6 depicts an overview of a call stack tied with tags. RegGuard creates a tag for each callee- and caller-saved region, where the tag of a caller-saved region also contains the previous tag of a callee-saved region or vice versa. This helps us to bind frames to each other for a tight representation of the whole program stack. Equations (6.1) and (6.2) below express what each tag created for caller- and callee-saved regions contains.

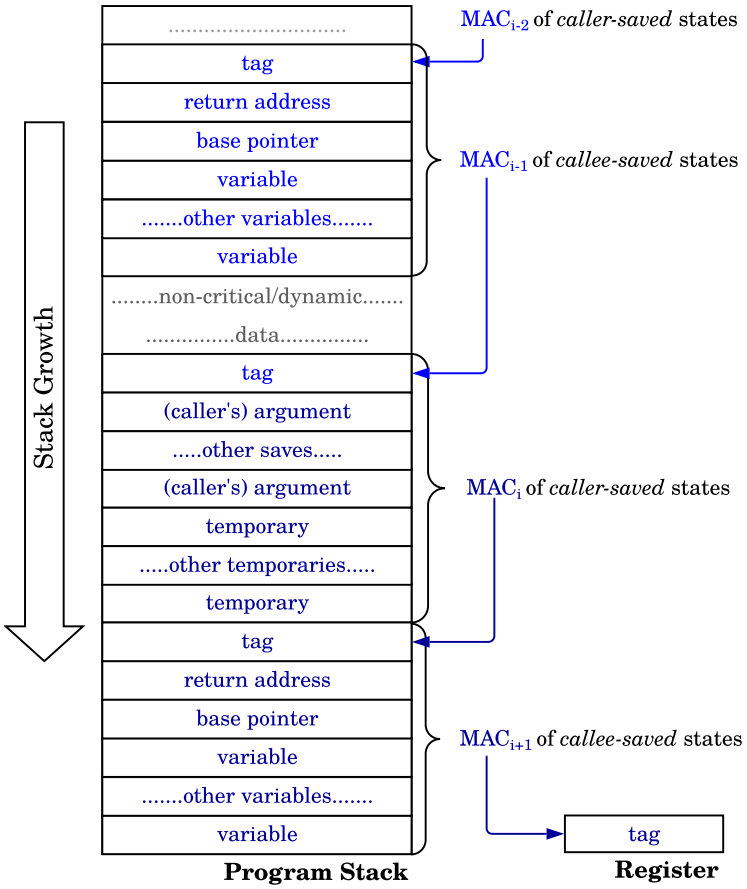


Figure 6.6: Securing saved register objects using a keyed hash.

$$\text{tag}_i = \text{MAC}_{\text{sk}}(\text{tag}_{i\text{-}1} \parallel \arg 1_{i-1} \parallel ... \parallel \arg n_{i-1} \parallel ... \parallel \text{tmp}n_{i-1}) \tag{6.1}$$

$$\text{tag}_{i+1} = \text{MAC}_{\text{sk}}(\text{tag}_i \parallel \text{ret}_i \parallel \text{bp}_i \parallel \text{var}1_i \parallel ... \parallel \text{var}n_i) \tag{6.2}$$

Although the details can vary depending on the calling convention and the architecture, we consider the caller is responsible for saving and restoring its arguments (*arg*) and temporaries (*tmp*) at call sites while its return address (*ret*), base/frame pointer (*bp*) and local variables (*var*) on registers saved by the callee. Even if the architecture (e.g., x86) does not use a link (return) register and stores the return address directly on the stack, it is still included in the tag generated for callee-saved regions as an object that must not be used until the return.

To reveal the corruption of a saved object, RegGuard injects two groups of instructions. The first group generates a reference tag for register values being saved

at function prologues and call sites. The second group checks whether this reference tag matches the one calculated from restored values. Both tag calculations directly align with existing register operations to avoid additional memory accesses. With a few scratch registers, RegGuard can compute tags from directly register values. In order to make this possible, the compiler rearranges register restores in the same order they are pushed, instead of pop instructions working in the reverse order.

6.3.2.3 Bootstrapping and Key Management

Regarding the bootstrapping of the system, the tag generation starts with the first call made by the software in question. For a simple setting with no process or privilege separation, such as a bare-metal or a RTOS environment, a single key to be shared by all tasks is generated at boot time using software or hardware RNGs available on the system. This key is assigned to an FPR dedicated as the key register. We note that this register is not saved to the memory by the scheduler or interrupt handler, thanks to the control over the software stack. If there is a hardware context switching in use, those instances also usually do not save FPRs. Otherwise, in the case of a general-purpose OS, a fresh key is generated at each process start. Only the kernel can save the key register to its own memory space, which is trusted by the user processes. User-managed threads share the same key and do not save the key register during a context switch.

6.3.2.4 Choice of MAC

The MAC function to be used should be chosen based on the available features of the CPU architecture. If the ISA provides relevant vector and cryptographic extensions, we recommend using HMAC-SHA256 with hardware acceleration. Otherwise, we suggest using SipHash [173] as an architecture-agnostic option for CPUs that lack cryptographic instructions. SipHash is a keyed hash primarily designed to be fast, even for short inputs, with a performance that can compete with non-cryptographic functions used by hash tables. Thanks to its performance benefits, SipHash is highly practical and deployable on different architectures.

```
read_buffer:                          #callee function
    .....prologue: register saves.....
    INIT(key)                         #initialise states (v1-4) with r_key
    store   r_tag,    [sp]
    COMPRESS(m1)                      #m1=r_tag
    store   r_ret,    [sp-8]
    COMPRESS(m1,m2)                   #m2=r_ret
    store   r_bp,     [sp-16]
    COMPRESS(m2,m3)                   #m3=r_bp
    store   r_var1,   [sp-24]
    COMPRESS(m3,m4)                   #m4=r_var1
    store   r_var2,   [sp-32]
    COMPRESS(m4,m5)                   #m5=r_var2
    r_tag = FINALIZE(m5)
    sub     sp,     40
    .....body instructions.....
    .....
    .....epilogue: register restores.....
    mov     r_tmp1, r_tag             #copy reference tag to a scratch register
    INIT(key)                         #initialise states (v1-4) with r_key
    load    r_tag,    [sp+32]
    COMPRESS(m1)                      #m1=r_tag
    load    r_ret,    [sp+24]
    COMPRESS(m1,m2)                   #m2=r_ret
    load    r_bp,     [sp+16]
    COMPRESS(m2,m3)                   #m3=r_bp
    load    r_var1,   [sp+8]
    COMPRESS(m3,m4)                   #m4=r_var1
    load    r_var2,   [sp]
    COMPRESS(m4,m5)                   #m5=r_var2
    r_tmp2 = FINALIZE(m6)
    CHECK(r_tmp1,r_tmp2)              #check whether the checksums match
    add     sp,     40
    ret
example():                            #code in Figure 6.5
    .....instructions.....
    mov     r_var1, &printf           #int (*func_ptr)...; line 2
    mov     r_var2, 0                 #int is_valid=0; line 3
    .....instructions.....
    call    read_buffer
    .....instructions.....
    mov     r_var2, 1                 #is_valid=1; line 12
    .....instructions.....
    cmp     r_var2, 1                 #if (is_valid==1) line 17
    .....instructions.....
    call    r_var1                    #(*func_ptr)(...); line 19
    .....instructions.....
```

Figure 6.7: MAC calculations aligned with register operations for the slice of func_ptr and is_valid variables.

Figure 6.7 sketches how our scheme aligns its MAC calculations with register operations at function prologues and epilogues using SipHash. Both sections start by initialising internal states (on scratch registers) generated from the key and constants. Next, it applies compression rounds on those with message blocks (values) already on registers. Lastly, it completes tag generation with the final message block (register). The reference tag is not pushed to the stack unless the function calls another function. Prior to the epilogue, this reference value is moved to a scratch register; the epilogue can thus restore the previous tag to the dedicated register as a part of the restoring process. The reference tag moved to a temporary register will be later compared against the actual tag generated from restored registers at the end before return. Any mismatch of two tags implies an attack because saved register objects cannot be changed unless the control is returned back to the caller function.

6.3.2.5 Attack Coverage

ROP attacks that exploit return addresses are prevented by RegGuard, regardless of whether the architecture has a link register or not as in x86. In contrast to other variable objects, return addresses are always static and must have a single definition (call) and single use (return) located at our instrumentation sites, so they are always included in the MACs and protected. Further JOP scenarios that alter forward-edge code pointers on the stack such as function pointers and switch statements are also mitigated as those are either securely updated (e.g. pointer arithmetic) within the CPU or checked against any corruptions before they are restored back from the stack. Thanks to the integrity guarantees on data pointers, absolute-address (non-linear) attacks that can use them to access/corrupt other memory sections are also avoided. In addition, our scheme mitigates relative-address (linear) attacks such that a stack array is overflown onto an adjacent condition variable as a DOP attack. We exclude scenarios that might alter composite data values such as strings for practicality. However, those strings typically host untrusted inputs and their corruption can be only meaningful as a data-only attack in case the given string has a critical use in bulk following a sanitisation check, with a timely bug located

between the sanitation and critical use. Otherwise, the sanitisation (comparison) outcome of those inputs that affects the control flow would be already transferred to a condition variable that will be safe on a register (i.e., control-flow bending attacks).

6.3.3 Security Analysis

As previously described, the adversary's goal is to manipulate the program runtime by corrupting control and data variables on the stack. We remind the reader that a critical variable is already safe from memory corruption while on registers. Therefore, the attacker can overwrite such a variable only if its register state is saved to the the memory because of another function call. For such corruption to stay undetected, the adversary has to either skip the MAC-based integrity checks or make those checks pass. We will look at each of these options in turn.

In order to skip checks, the adversary must modify the binary or its execution to void the instrumentation. The former is not possible in our model because the code segment is non-writable. The latter, which requires altering code pointers, is also infeasible as the scheme protects those in the first place. For the adversary to pass integrity checks, he has to forge a valid tag or reuse a previously recorded one. Forging a valid keyed hash for an attack state either requires finding the second preimage of the legitimate state or access to the key. Since the key is protected on a register that is never saved to the same address space (including user-managed context switches and setjmp/longjmp instances), and therefore unavailable to the attacker, if the MAC-function is secure (i.e. existentially unforgable, and second preimage resistant), forging a valid tag without the key is only possible with negligible probability. We remind the reader that our system model assumes that the software code executing within the same process/program space, including user libraries, are recompiled or modified through binary-level replacements to guarantee that no instruction operates on the register (i.e., FPR) reserved for the key, except bootstrapping code responsible for key generation and placement. This is required to ensure the confidentiality of the key even under the presence of a powerful attacker that has arbitrary access primitives to the same program memory. Otherwise, in

Table 6.2: The details of calling convention used.

Register	Type	Purpose
x0-x7	Caller-saved	Arguments
x9-x15	Caller/e-saved	Temporaries
x19	Callee-saved	Tag
x20-28	Callee-saved	Local variables
x29	Callee-saved	Frame/base pointers
x30	Callee-saved	Link/return addresses
q31 (FPR)	Reserved/not saved	MAC key

the case of a multi-threaded environment, the key register is allowed to be managed (i.e., context switches) by only trusted software components, such as the kernel.

The adversary might attempt to replay a seen tag for a different call of the same or a different function. However, even with the same variable and argument values, replaying will not work. This is because each tag containing return address, base pointer and more importantly former tags (representing previous call frames) provides a very tight representation of the whole stack, where the (most recent) tag digesting all context is also safe on a register. Besides, replaying a tag for a different process in rich OS environments or a different execution time in the embedded systems is not an option since a fresh key is generated at each process or device start.

6.4 Implementation on ARM64

We have implemented a proof-of-concept[1] of RegGuard on ARM64 (AArch64) to evaluate its performance impact. RegGuard can be adapted to different architectures such as x86, SPARC, MIPS, PowerPC and RISC-V (preferably 64-bit versions). But we have chosen ARM64 for demonstration purposes due to the following reasons:

ARM has been the dominant architecture of the mobile and embedded landscape for a long time. Thanks to Apple's recently started transition to ARM-based processors and the embrace of Microsoft Windows, it is now projected that ARM will surpass Intel in the PC market in less than a decade [174]. Apart from promising market share, ARM64, with 31 GPRs (64-bit) and 32 FPRs (128-bit), has more

[1] https://github.com/msgeden/llvm-project

registers than x64 (i.e., 16 GPRs and 16/32 FPRs). Hence, even without having to modify the standard calling convention (ABI) of underlying software components, ARM64 provides enough registers to secure more variables than expected to be found per function (see Figure 6.1). For instance, the standard ABI dedicates 10 callee-saved registers compared to 6.9 variables found on average. Furthermore, registers reserved for arguments and temporaries not only help to secure other potential targets but also avoid register pressure in general. It also enables to use a FPR as two GPRs via vector form indexes. Besides, the ISA equipped with cryptographic extensions allows us to evaluate the hardware-accelerated HMAC-SHA256 option.

For the implementation, which consists of two parts, we have used the LLVM compiler, which is configured to dedicate a single FPR (128-bit) for the key and a GPR (64-bit) for tag values. For the first part, we have modified the *basic* register allocation pass provided as a custom technique using priority queues to eliminate strict visits in linear order. Since the benchmark programs have not encountered with register pressure, our allocation pass simply ensures that registers given to variables are not spilled for performance reasons. For the second phase, we have mainly worked on the part responsible for target-specific prologue and epilogue code. For the proof-of-concept, integrity checks are placed for only callee-saved registers that are primarily assigned to local variables by the allocator. But the registers known as caller-saved can also be included in tag calculations using the same instrumentation, thanks to the compilation flags available (e.g., `-fcall-saved-x9`). Table 6.2 summarises the highlights of the calling convention used during our experiments.

For simplicity, we have encapsulated MAC calculations with two functions added to the C library in use[2]. The first one (`__register_mac`) is injected to the end of the prologue and generates a reference tag from saved register values. The second one (`__register_check`), which is placed at the beginning of the epilogue, creates another tag from the values to be restored and compares it against the reference value. In the case of unmatched values, which means an attack, it terminates the program. Both wrapper functions take the start address and the size of the region where

[2]https://github.com/msgeden/musl

registers are pushed as their arguments. The latter function additionally requires the reference tag for comparison. The instrumentation also handles the preservation of original arguments required by the actual callee function and the return values upon its completion at call sites of the wrapper functions. For optimisation purposes, we have avoided injecting these functions to the leaf functions of the program as their frames cannot be modified in practice without another function call.

Differently from the ideal design proposed in Section 6.3.2, those wrapper functions calculate MACs from register values awaiting on the stack instead of directly using values already on registers. We remind the reader that as a proof-of-concept implementation avoiding the complexity, these wrapper functions introduce additional cache hits. Hence, our performance discussion should be seen as an over-approximation, whereas a production-ready implementation based on the proposed design would have less performance overhead.

For MAC, we have implemented two options. The first option is HMAC-SHA256, backed by hardware acceleration. The second one is SipHash-2-4 producing a 64-bit tag, as a fast, practical, and deployable option for different architectures lacking advanced vector and cryptographic extensions.

6.5 Evaluation

This section first approximates the performance overhead of RegGuard using our proof-of-concept (PoC) implementation. Then, it examines its security promises against real-world vulnerabilities.

6.5.1 Performance

For performance evaluation, we have used *cBench* [175] , a popular open-source uniprocessor benchmark suite that is based on earlier MiBench [158] suite. The experiments were performed with a collection of 14 C programs from various categories that aim a realistic benchmarking and research. We have run those programs on a Linux system running on an Apple M1 chip that is equipped with the ISA features we need, such as SHA extensions. We have instrumented not

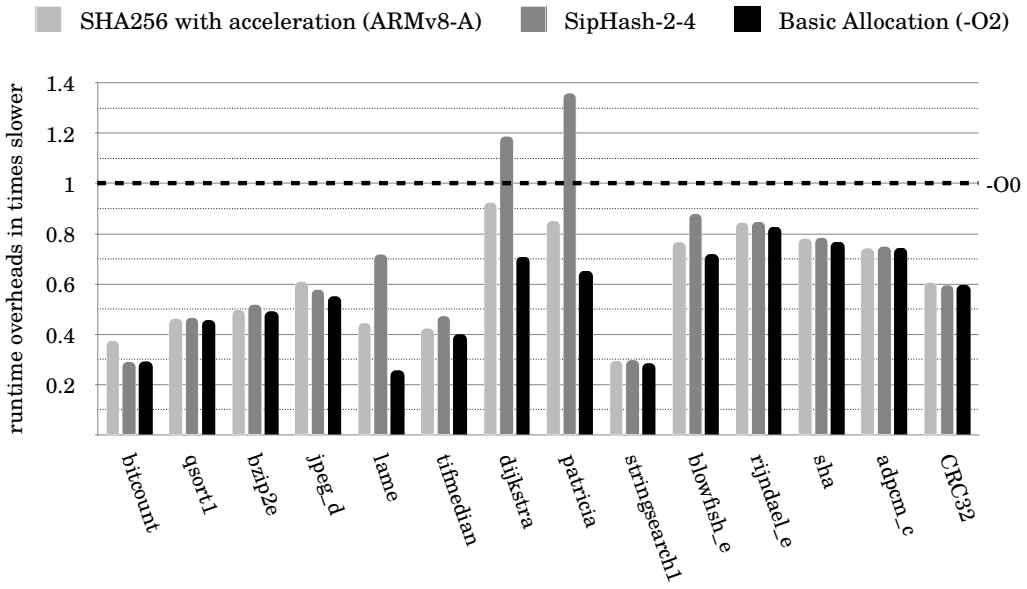


Figure 6.8: Runtime overheads of program-only instrumentation.

only benchmark executables but also the C library interacting with the kernel to have a better understanding of performance costs in the case of extended guarantees. The full instrumentation of the C library aims to mitigate scenarios where the libc vulnerabilities can be exploited to corrupt the program's stack objects. We have experimented with both SHA256 (using ISA acceleration) and SipHash-2-4 for integrity checks.

6.5.1.1 Program-only Instrumentation

In the case that only the program binaries are instrumented, both MAC implementations promise better execution times compared to unoptimised binaries (-O0), where no register allocation takes place. As seen in Figure 6.8, only two benchmark programs with SipHash have produced slower execution times than unoptimised versions. Considering a comparison between the basic register allocation without any instrumentation and our scheme compiled with the same optimisation level (-O2), SHA256 backed by native ARMv8-A instructions has produced only *13%*, whereas SipHash yields *23%* overhead.

6.5.1.2 With C Library Instrumentation

We have observed higher performance costs for programs linked to an instrumented C library as expected. Compared to the naive scenario where both benchmark programs and libc are neither instrumented nor optimised, our implementation has still produced better execution times on average for the suite. Only three programs using HMAC-SHA256 and four programs with SipHash out of 14 benchmark executables have had slower execution times than non-optimised versions. In contrast to the basic register allocation bundled with -O2 optimisations, SHA256 and SipHash instrumentation have introduced *33%* and *59%* runtime overheads, respectively. Considering the binary sizes, instrumented C library with wrapper functions is only *14%* higher than the non-instrumented library file.

Because the optimisation configurations do not allow us to measure the performance impact of register allocations in isolation, we have used -O2 as the default optimisation level. Comparisons with basic register allocation create a baseline scenario to understand the standalone costs of additional integrity checks. On the other hand, experiments with unoptimised and non-instrumented programs aim to measure the compensation level by the register allocations of our scheme. We note that there are other optimisations included contributing to the overhead compensation. For instance, inlining some functions not only avoids branching costs but also reduces tag calculations. This is due to the fact that the caller can aggregate register operations of the inlined function. Overall, SipHash, with its reasonable overheads, proves to be a practical option for different CPU architectures without asking for any hardware change or acceleration. If available, similar to ARMv8.3-A, using native SHA instructions that provide around *7x* speed-up would be a faster and more convenient option. Depending on the CPU features, both options can thus be practically used to ensure the integrity of register data on the stack since the overheads are within very small fractions of optimised times (-O2) for most programs.

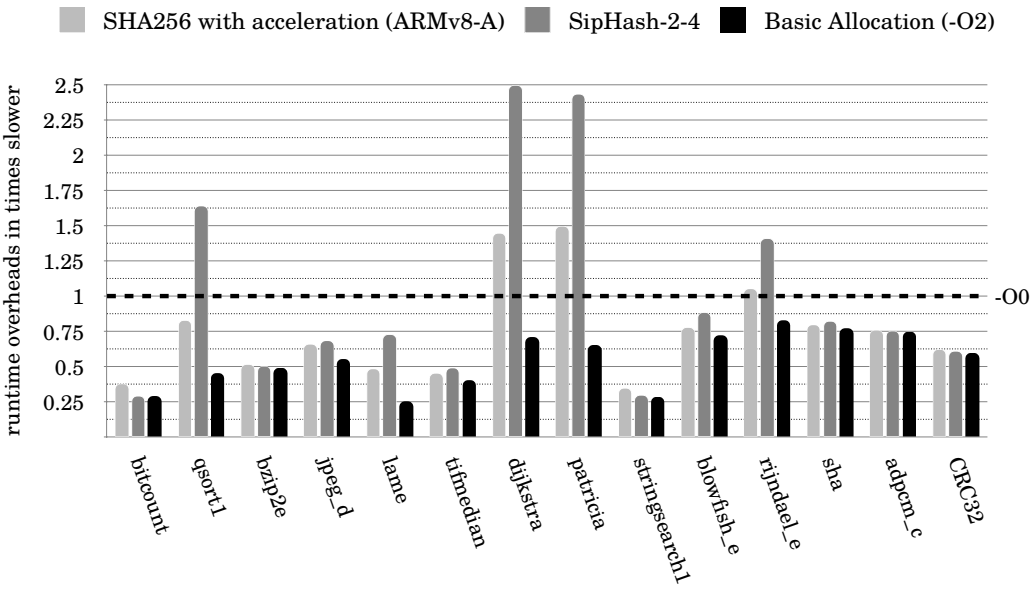


Figure 6.9: Runtime overheads with libc instrumentation.

6.5.2 Security and Real-World Cases

Our PoC implementation in ARM64 accommodates the local variables of tested programs in callee-saved GPRs, where the architecture provides plenty of, therefore, ensures that those variables are safe from corruption while in registers during the execution of a function. Whenever those variables are saved to the stack because of another function call, PoC instrumentation checks whether those variables, return addresses (in the link register), and frame pointers are restored to the registers without any modification. Unlike mitigations whose verification data must be isolated in the same memory space, our cryptographic approach also provides protections against corruptions that can be achieved through different means, e.g., rowhammer attacks, as long as the key(s) are kept confidential. In such cases, the attacker gains only a brute-force attempt to target the underlying MAC primitive.

We have tested our PoC implementation using buffer overflow cases extracted from open source model programs (e.g. BIND, Sendmail, WU-FTP) made available as a SARD test suite (88) by NIST. For a sound evaluation, we have first attempted to compile available 14 cases with clang and -O2 flag instead of the default gcc and -O0 configurations given by the suite. Due to the optimisations changing

the memory layouts and architectural differences, solely 6 cases have remained compiled and exploitable. From those, our implementation has successfully captured 5 out of 6 cases (1285/CVE-1999-0368, 1287/CVE-1999-0878, 1289/CA-2001-01, 1299/CVE-1999-0131, 1303/CVE-1999-0047). Only one case (1307/CVE-2001-0653) that exploits a sign-cast bug to underflow a global array with negative index values has been undetected. Although such scenarios are not within the scope of this work, Section 6.6 discusses how our MAC checks can be extended to mitigate similar corruptions.

6.6 Discussion

In this section, we present a discussion of certain design decisions of RegGuard, including further extensions and future CPU design features that would complement our scheme.

6.6.1 Chained vs Independent Frames

Given that our scheme uses a keyed hash, it is not a strict requirement to include the previous tag in the tag of the next frame. In other words, we could have chosen to independently secure each call frame, rather than chaining them together. This section will briefly look at the reasons for and against this design desertion.

For a program with a call stack strictly following LIFO, we could have relied solely on a single (unkeyed) hash for the stack *integrity* by chaining frames. This is because such a program can ensure that any CPU state restored from the stack complies with the hash register first. However, there are many legitimate cases where the register hosting the head of the chain has to be saved to/restored from program memory without our instrumentation, for example, setjmp/longjmp, exception handling and user-managed threading instances. They all oblige us to rely on the MAC key instead of a single hash.

Despite its redundancy for integrity assurance, we have chosen the chained approach over independent frames to prevent *replay attack* scenarios. With independent frames, the attacker can simply replay a call frame (and its aligned

tag) for a different function call or context. However, with a chained approach, replaying for a different call context will not work since the tag register provides a very tight representation of the execution context, including all functions calls waiting to be returned. Even though setjmp/longjmp and user-managed thread instances might still provide a small window, it is very unlikely for the attacker to find a useful tag he can replay. This is because he needs a more coarse-grained stack-size image this time. Also, he will have fewer options; for example, he can exploit only setjmp/longjmp instances instead of function prologues/epilogues.

The only downside of a chained approach is occupying an additional register, which has to be excluded from allocations. This might be an issue for some legacy or primitive architectures that suffer from register scarcity. In such cases, the independent frame approach can be preferred to avoid the use of an extra (tag) register. To harden replay attacks without chained frames, we suggest including the stack pointer and a static function identifier or a nonce generated by the compiler as an immediate value in tag calculations. These two parameters provide a good approximation of the context by describing the current stack depth and returning function. The attacker cannot modify the function identifier, thanks to the code integrity. Also, the stack pointer would be safe by default on a register that can be saved to memory for the same reasons as the tag register.

6.6.2 Primitive Devices and Register Scarcity

Our proposal uses security scores to distinguish critical variables and prioritise them for available registers under register pressure scenarios. However, it is difficult to observe such use cases with a modern CPU core providing a register file consisting of 48-64 (16/32 GPRs and 32 FPRs) registers with sizes up to 2kB. Hence, our selection process actually serves more primitive architectures suffering from register scarcity (e.g., 6-8 GPRs with no FPRs). In such a case, our security scores aim to accommodate at least all critical objects in the registers. But if there is still a critical object (e.g., condition variable) left in the memory, the compiler can display a warning; so the programmer would review the code. Despite being ignored by

some compilers, the programmer can use the register keyword in C to annotate which variables to protect. A different approach for CPUs with register scarcity can be to adapt RegGuard as a local register allocation scheme. Such a scheme would mitigate the register pressure problem by enabling the reuse of registers at a smaller (basic block) level in return for higher overhead.

We have designed RegGuard as an architecture-agnostic solution to make it applicable to a wide range of systems, even with the most resource-constrained devices in mind; for example, a 16/32-bit MCU with no security at all, but might still be prevalent in critical systems. By relying solely on flash program memory and a few GPRs, we can significantly reduce the attack surface with shorter keys and checksums against less strong adversaries.

6.6.3 Future CPU Architectures

Although RegGuard is designed to fit existing CPU architectures, we would like to see CPU manufacturers incorporate some of these ideas into their designs in the future.

If the next generation of CPUs were to include the necessary registers and maybe even hardware acceleration of a suitable MAC function, RegGuard could be implemented at the hardware level through a single instruction. A bit vector-like operand can be given to describe which registers to include in the MAC, and the new instruction can then run all the necessary calculations within the CPU. Such instruction would enable us to create a standalone tag for each spilled value due to register pressure within the same function, without having to worry about performance overheads.

Furthermore, similar to the Itanium (IA-64) architecture providing 128 GPRs and 128 FPRs, CPU manufacturers can consider expanding their register files as trusted storage and adopting *register windows* to zero out the performance costs in return for space overhead within the CPU. Register windows, which are designed to avoid the cost of spilled registers on each call by making only a portion of registers visible to the program, can actually benefit our scheme more than its original purpose by eliminating cryptographic calculations. For example, with a

window size of 32 (from 128 registers), RegGuard would not incur any overheads for a program that has no call down deeper than four calls.

6.6.4 Further Extensions

RegGuard covers attack scenarios that require modifying a stack object in the first place. Due to the integrity assurance of pointers on the stack, most illegitimate accesses to other memory sections outside the stack would also be mitigated. However, there might still be some options for the attacker not addressed by the protection of stack pointers, such as overflowing a global array or a heap buffer to target an adjacent variable whose modification can result in a successful attack. But thanks to the key register and MAC properties, we can extend our scheme to ensure the integrity of these objects. For example, we can allocate a tag address next to each global variable or composite data that will host a digest of them. We can update this tag at each legitimate (re)definition of those variables and verify when used.

6.7 Summary

This chapter presented a novel and practical runtime protection scheme, called RegGuard, which leverages CPU registers to mitigate stack-based control- and data-oriented attacks, which exploit memory vulnerabilities commonly found programs developed by C-like unsafe languages. Our proposal relies on the immunity of registers from memory corruptions as unaddressable storage units. Despite their heavy use by compiler optimisations, CPU registers have not been systemically used as secure storage because of their limited capacity and voided immunity of saved values on the stack. This chapter addresses these issues with a two-step proposal. First, during register allocations, it suggests favouring variables that are more likely to be targeted; so they stay safe while in use. Second, when those registers are saved to the stack because of a new function call, our scheme computes a keyed hash to ensure that they are restored back to registers without any corruption. These integrity checks enable the reuse of the register file as secure storage repeatedly for each function call, without having to occupy registers across function boundaries.

Although the solution given in this chapter is designed as a software-based approach to be practical, it makes strong security promises using a very basic hardware primitive, i.e., CPU registers. This makes our approach applicable to a very broad range of devices, from high-end to low-end, without asking for special hardware features. Our experiments on ARM64 showed that register allocations can improve both security and performance together with a surplus within the range of 13% (with SHA extensions) to 23% (SipHash) on average compared to purely performance-based optimisations. The chapter proposed the first systematic use of general-purpose CPU registers for security purposes. It presented a runtime protection scheme relying on building blocks that are available in most computers, such as code integrity, registers, and MAC calculations that can be expressed by any CPU ISA.

"I think computer viruses should count as life...I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image."

— Stephen Hawking

.

7

Using Runtime Features for Identification of Malicious Software

Even if the solutions delivered in previous chapters can ensure the runtime integrity of vulnerable software programs and can reduce their attack surfaces meaningfully, there would still be many ways to compromise a target system. For instance, a device owner or an operator can be tricked to install malware on a system, as a consequence of a phishing or social engineering. This chapter considers such a setting where the host system is already infected by sophisticated malware. Therefore, it presents a malware analysis framework for the evaluation of system-level runtime features that can be used to identify malware executions. Unlike previous chapters that focus on the protection against malicious executions of legitimate but vulnerable software instances, this chapter investigates the runtime of programs that are malicious by design, and whose code or static analysis might not be an option due to the evasion means in place (e.g., polymorphic malware).

7.1 Introduction

Malware mostly refers to a software instance or executable that is designed to gain unauthorised access, steal data, or demand a ransom. Malware can exploit software vulnerabilities, which we aimed to mitigate in previous chapters, as the entry point

to infect host systems. Depending on the purpose and propagation mechanism, they can be categorised as virus, trojan, worm or ransomware. Despite the widespread use of antivirus (AV) solutions for a long time, new malware variants continue rising to take control of our computers, smartphones, and IoT systems.

On the other hand, malware detection and classification are generally considered as a feature or signature extraction problem, from known (labelled) malware samples to seek similarity with the unknown (non-labelled) ones based on selected features. There have been many proposals that rely on static methods, such as extracting opcode [137] or CFG-based [176] signatures from the binaries. Despite their advantages in code coverage, static methods can suffer from evasion techniques that obfuscate or encrypt the malicious code (i.e., polymorphic malware [177]). Fortunately, dynamic methods using runtime features can overcome most of these evasion strategies, as most malware instances eventually execute on the target system. Although there are many studies proposing various runtime features, there is a lack of comprehensive work that evaluates the value of choosing a feature model over others. Therefore, this chapter presents an experimental exploration of different runtime features, in order to contextualise and evaluate existing knowledge. For this purpose, it implements a malware analysis framework that combines ML and sandbox methods to distinguish malware families from the same category (i.e., ransomware), which is chosen as a more challenging task, compared to simpler detection or category-based classifications.

The rest of the chapter is organised as follows. Section 7.2 defines the research questions that we intend to address through this work. Section 7.3 describes the design of the framework. Finally, Section 7.4 evaluates and discusses the results of the experiments.

7.2 Problem Definition

This chapter aims to evaluate and contextualise existing knowledge on the use of runtime features for malware identification. Briefly, we have collected malware samples from different ransomware families and have used an open-source automated

malware analysis environment, called Cuckoo sandbox, to record their runtime interactions with the host system. This is to investigate the use of different feature representations that are extracted from recorded call traces, such as bag-of-words, ngrams, and wildcard based search patterns; and other forensic type features such as files accessed and registry keys available in the generated runtime reports. This chapter aims to answer the following research questions:

- **RQ1: Can API (and system) call traces be leveraged to classify malware families from the same category? (Ransomware)**. This work uses API and system call traces as the main runtime feature source for the classification of malware families from the same category. During the sample collection, we picked only ransomware instances, in order to minimise potential biases that might arise from different categories. This aims to create sound experiments because we should still expect all the ransomware instances to behave similarly at high level, such as encryption of the files and displaying payment instructions for the ransom.

- **RQ2: Which feature models extracted from API and system calls perform better for malware classification?** By experimenting with different feature representations of call traces, such as bag-of-words, ngrams, wildcard search models, we explore the drawbacks and advantages of each, regarding accuracy, scalability, and potential resiliency to evading techniques.

- **RQ3: To what extent can other runtime artefacts on the system be used to identify malware families?** Apart from call traces, the framework also explores other runtime artefacts available in the Cuckoo reports, such as the files accessed, registry keys obtained, mutexes created and DLL files loaded. Although the call traces provide the same information as part of the function arguments, we intend to understand the value of those artefacts without the noise of other calls or arguments.

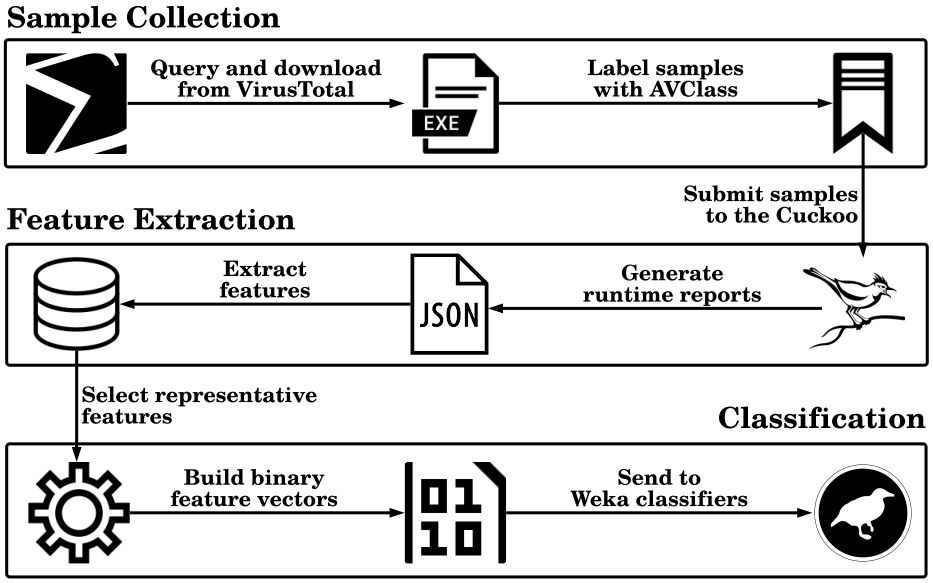


Figure 7.1: The overview of malware identification framework.

- **RQ4: Which techniques yield better results in terms of feature selection and classifier algorithm?** Since some extractions generate more features than the classifiers can handle, we should employ only the most important ones. This requires a feature selection method, which is an important factor that can affect classifier outcomes. Additionally, different classifier algorithms can produce dissimilar results. Using heuristics, we aim to understand optimal settings that provide better results.

7.3 Methodology

As shown in Figure 7.1, our experimental framework[1] consists of three main stages: *1-Sample Collection*, *2-Feature Extraction* and *3-Classification*. The framework first collects labelled samples from different ransome families. Then, it generates runtime report for each using Cuckoo sandbox and extracts different features from these reports. After the feature selection phase, malware samples are represented as binary feature vectors that are sent to Weka [178] classifiers.

[1]https://github.com/msgeden/familyclassifier

Table 7.1: Number of malware samples used from each ransomware family in training and test portions.

Family	Training	Test
Cerber	48	24
Crysis	47	23
HydraCrypt	24	11
WannaCry	39	19
Total	159	77

7.3.1 Dataset Collection

The samples are selected from four different popular ransomware families *cerber, crysis, hydracrypt* and *wannacry*, which we have found the highest number of labelled samples compared to the other options [179]. We queried VirusTotal [180] for portable executable (exe) files using those family names as query keywords. Our queries were also filtered by the conditions of being detected by at least five antivirus engines and having a sample size of less than 5MB. Despite queries based on family keywords, each antivirus vendor can still label a given sample with a different family name. Since VirusTotal returns any sample that is labelled by at least one of the 82 AV engines with the keyword, the query results do not provide a strong indication of family belongings. To address this problem, there are solutions [181, 182] that extract the most popular family name from AV labels in VirusTotal reports. Therefore, we eliminated misleading samples whose labels are different from queried keywords by using the AVClass tool [182]. In other words, we only used the samples if the label given by this tool matches with the family name queried.

Following the sample collection, we split the dataset into training and test portions with a 2:1 ratio, as the most common ratio used in ML-based malware detections. We used separate training and test portions instead of cross-validation methods. This is because integration of the feature selection phase with cross-validation requires extra effort to avoid overfitting bias during classifications [183]. The number of samples used in each portion can be seen in Table 7.1.

```
GetFileType(0)                                  /*Assigned ID:A1*/
NtClose(0xb0)                                   /*Assigned ID:B2*/
RegCloseKey(0xa4)                               /*Assigned ID:C3*/
NtTerminateProcess(0,0,1)                       /*Assigned ID:D4*/
```

Figure 7.2: A short call trace example.

```
GetFileType,NtClose..                                  /*API calls*/
GetFileType(0),NtClose(0xb0)..                         /*API calls with args*/
GetFileTypeNtClose,NtCloseRegCloseKey..                /*API calls 2-grams*/
A1[0-9A-Z-]0,12B2,B2[0-9A-Z-]0,12C3..    *API wildcards (2/4-junk calls)*/
NtClose,NtTerminateProcess..                           /*System calls*/
NtClose(0xb0),NtTerminateProcess(0,0,1)..              /*System calls with arg*/
NtCloseNtTerminateProcess..                            /*System calls 2-grams*/
B2[0-9A-Z-]0,12D4..                  /*System wildcards (2/4-junk calls)*/
```

Figure 7.3: Different feature representations that are extracted from the sample call trace given in Figure 7.2.

7.3.2 Runtime Data Generation

After the collection and labelling phase, we ran the samples within the Cuckoo sandbox analysis environment. Cuckoo collects runtime reports from running VM guests. It allows monitoring API and system calls, the files accessed, registry keys and mutexes used by the submitted sample, and collects network traffic for further analysis. Using *Windows XP-SP3* as the target system, we ran each malware sample for 120 seconds. For each submission, Cuckoo generated a JSON report that contains the runtime artefacts used for our feature extractions. The detailed structure of these reports can be found in Figures A.5, A.4 and A.6 in the Appendix.

7.3.2.1 Feature Models

We have used call traces, the file names accessed, registry keys and mutexes as runtime features. We put special effort into raw call traces by extracting different representation models. The list of experimented feature models with the number of unique features extracted from each can be found in Table 7.2.

Call Traces. Windows's API and its subset system calls can be used in different ways to identify malware samples. This chapter compares the accuracy and scalability of these options using different representations of API and system calls.

The simplest feature model that we have derived from both API calls and system calls is the *bag-of-words* representation without function arguments. On the other hand, the bag-of-words with arguments option generates the highest number of unique features (see Table 7.2). Those arguments provide detailed information about the intention of the call made. Unlike bags-of-words, *ngram*-based models should give more meaningful features that can have a better corresponding to the malicious behaviour. Although ngram models with function arguments are not computationally feasible due to the huge feature space, we have successfully generated naive 2-grams and 3-grams of both API calls and system calls.

Since malware with sophisticated evasion mechanisms can defeat ngram models by injecting junk calls that break the expected sequence, this chapter also explores the use of wildcard-based search models on the traces. These search patterns intend to capture the corresponding malicious action, where the order of calls can be more indicative, regardless of the ngram sequence. For an efficient implementation of this model, which adds an extra layer of complexity, we first assigned base-36 ID numbers to each function name to minimise the search cost on the traces. Then, we created regular expressions as the features to be searched for all possible permutations of the required size (2-calls, 3-calls) with an adjustable distance buffer by using the function IDs and wildcard characters. We set the distance buffer as four junk calls since larger distances can result in false positives with unnecessary search costs.

We extracted ngrams and wildcard patterns for 2-calls, 3-calls without arguments. Function arguments are only used with the bag-of-words model. To illustrate all these call trace-based models, a short fabricated call trace and the feature models extracted from this trace are given in Figures 7.2 and 7.3.

Files. A forensic type feature, the files accessed during a malware execution can be an indicator of its family. In addition to opening and reading, malware samples can also create new files or write to existing ones. This framework explores the names of files accessed as a usable feature to identity malware families.

Registry Keys. In Windows systems, malware execution can leave important footprints in the registry database. Although there is not much research on a registry-based malware, it is known that are instances [184] in the wild that reside only in the registry without causing any file artefacts, which can be challenging for antivirus programs.

Mutexes. Mutex objects in operating systems represent the synchronization objects that manage the shared resources by different threads. Since a malware program can also use shared resources, this work explores the use of mutex objects as a runtime feature as well [185].

Dynamic-Link Libraries (DLL). Another feature type investigated is DLL files loaded during malware executions. Although this information can give a hint about the API functions called, we have experimented with DLL names, to understand to what extent those coarse-grained features are usable.

7.3.3 Feature Selection

We hav explored different feature selection techniques to eliminate non-informative features for classifiers as some models (e.g., API calls with function arguments) extract more features than the classifiers can handle (see Table 7.2). Most malware studies use *Information Gain* [137, 142, 186], whereas *Document Frequency Threshold* [139] or *Fisher Score* and *Chi-Square* scores [187, 188] are other popular options used to decide on the features to be used.

In addition to our experiments using *Information Gain*, we adapted a new feature selection method that was previously used for binary malware detection problems to our multiclass family identification problem. This method is called

Table 7.2: Number of unique features extracted from dataset for each feature model.

Feature Model	# of Features
API Calls	210
API Calls 2-grams	2690
API Calls 3-grams	10823
API Calls with args	1907098
API Calls wildcard 2-calls	6635 / 45769
System calls	35
System calls 2-grams	321
System calls 3-grams	1396
System calls with args	958474
Sys calls w.card 2-calls	651 / 1369
System calls wildcard 3-calls	6748 / 50653
Files accessed	55581
Dll loaded	166
Registry keys	4785
Mutexes	101

Note: Wildcards models (A/B) represent the features found (A) and permutations generated (B).

Normalised Angular Distance [189]. Because these selection methods can result in feature sets dominated by specific families, the framework also offers classwise adaptations to obtain a fair representation of each family in the selected feature set.

7.3.3.1 Information Gain

Information gain method (also called average mutual information by Yang et al. [138]) was first used by Kolter et al. [137] for the selection of ngram bytes. To calculate the information gain score of a feature f by using Equation 7.1:

$$\mathrm{IG}(f) = \sum_{X \in \{f, \bar{f}\}} \sum_{C_i} P(X, C_i) \log \frac{P(X, C_i)}{P(X)P(C_i)} \tag{7.1}$$

the term C_i represents the class of families, while the terms f and $\bar{f}$ represent the existence or absence of the given feature. Thus, $P(X, C_i)$ can be expressed as the proportion of existence or absence of the feature f for the given family C_i, while $P(X)$ represents the proportion of samples in the whole set that contains or does not contain feature f, and $P(C_i)$ is the proportion of samples that belongs to the given class C_i in the whole collection set.

7.3.3.2 Normalised Angular Distance (NAD)

A new feature selection technique [189], NAD leverages the representation of features in a vector space where each dimension corresponds to the class likelihoods of the features expressed as $P(f|C_i)$ and defined as the proportion of samples that contain the feature f for the given family class C_i.

This method relies on the assumption that feature vectors that have equal class likelihoods for each class are not distinguishing and should not be selected. As the distinguishing power of feature increases, the ratio of the difference between class likelihoods should increase as well, where our approach measures it via angular distance between the feature vector and reference vector that has equal class-likelihoods.

After representing all features as vectors in the probability space, using Equation 7.2, the method first calculates α the angular distance (in radians) between the feature vector $\vec{f}$ and any reference vector that has equal likelihoods for all classes such as $\vec{r} = (1, 1, 1, 1)$.

$$\alpha = \cos^{-1} \frac{\vec{f} \cdot \vec{r}}{||\vec{f}|| \cdot ||\vec{r}||} \tag{7.2}$$

However, regardless of the vector magnitudes, this angle will be the same for the features that have the same likelihood ratios such as $\vec{f_1} = (0.1, 0.2, 0.3, 0.4)$ and $\vec{f_2} = (0.01, 0.02, 0.03, 0.04)$ which can result in the selection of noisy and sparse features.

In order to manage this trade-off between being more common and more distinctive, NAD takes the magnitude of vector into account, as the normalisation factor, with a degree parameter k that can adjust the importance of the magnitude, for the final score. During our experiments, we set $k = 2$, which could have been experimented within a range of $[1.5, 4]$.

$$\text{NAD}(f) = \alpha \times ||\vec{f}||^{1/k} \tag{7.3}$$

7.3.3.3 Classwise Selections

To prevent the feature sets from being dominated by specific families, and to provide a fair representation of each class, we have modified the scores of *Information Gain* and *Normalised Angular Distance* in a class-wise fashion.

Although previous studies [139, 190] propose classwise feature selection in different ways to solve the issue, we offer a more practical solution. We first create separate ranked lists for each family class using Equations 7.4 and 7.5 that score only the features that have the highest class likelihoods for the given class of the list. We then build our final set with the features ranked in each class list by ensuring that for every n number of features there will be $n/|C|$ features from each family class to create an equally distributed feature set for the classifiers.

$$\text{CWIG}(f, C) = \begin{cases} \text{IG}(f), & \text{if } C = argmax_{C_i} P(f|C_i) \\ 0, & \text{otherwise} \end{cases} \tag{7.4}$$

$$\text{CWNAD}(f, C) = \begin{cases} \text{NAD}(f), & \text{if } C = argmax_{C_i} P(f|C_i) \\ 0, & \text{otherwise} \end{cases} \tag{7.5}$$

7.3.4 Classifications

Following the extraction of different feature representations from Cuckoo reports, we applied a feature selection phase for models that produce more than a thousand unique items (see Table 7.2). Then, we created binary feature vectors with a length of 1000 for each sample where the ones represent the existence of a feature for the given sample and the zeros describe the absence of that feature. Those sample vectors, which constitute matrices for each training and test portion, were later sent to the Weka [178] classifiers. For classifications, we experimented with *k-Nearest Neighbour* (k=3), *Support Vector Machines* (with SMO functions and poly kernel), *Random Forests* (# of trees=100) and *Neural Networks* (with default settings of Multilayer Perceptron provided by the beta package).

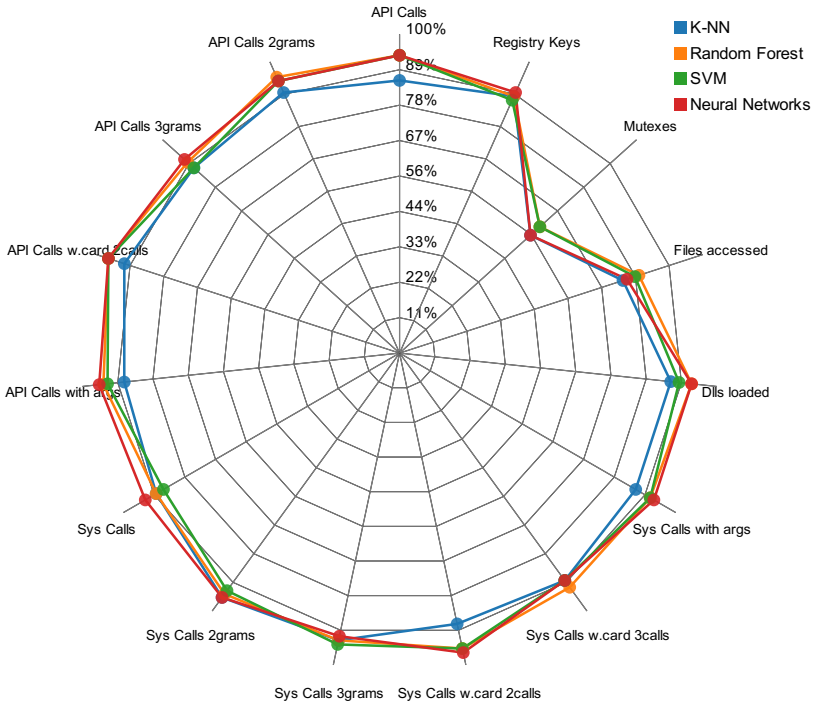


Figure 7.4: Weighted TPRs of all feature models for different classifiers (Classwise NAD used).

7.4 Results and Discussion

We experimented with 16 different feature models, 4 feature selection techniques, and 4 classifier algorithms on a dataset consisting of 236 samples. Our experiments yielded promising results. Although we consider correct classification ratio as the key performance metric for our discussion, we will evaluate the scalability of different feature models. Correct classification ratios can also be defined as classification accuracy or weighted true positive ratio (TPR) of all classes as the terms that we will use interchangeably.

Because of many combinations caused by different experiment settings, such as feature models, selection techniques and classifier algorithms, we will discuss the results with the settings that yield better results on average, which are Classwise NAD (selection technique) and Neural Networks (classifier) (see Tables 7.3 and 7.4).

Table 7.3: Accuracy Results for Classifiers with Classwise NAD.

Feature Model	K-NN	RF	SVM	NN	Average
API Calls	85.53%	93.42%	93.42%	93.42%	91.45%
API Calls 2grams	89.47%	94.74%	93.42%	93.42%	92.76%
API Calls w.card 2calls	90.79%	96.05%	96.05%	96.05%	94.74%
API Calls 3grams	86.84%	89.47%	86.84%	90.79%	88.49%
API Calls with args	86.84%	93.42%	92.11%	94.74%	91.78%
Sys. Calls	88.16%	88.16%	85.53%	92.11%	88.49%
Sys Calls 2grams	94.74%	93.42%	92.11%	94.74%	93.75%
Sys Calls w.card 2calls	86.84%	94.74%	94.74%	96.05%	93.09%
Sys Calls 3grams	92.11%	92.11%	93.42%	90.79%	92.11%
Sys Calls w.card 3calls	88.16%	90.79%	88.16%	88.16%	88.82%
Sys Calls with args	85.53%	90.79%	90.79%	92.11%	89.81%
DLLs loaded	85.53%	92.11%	88.16%	92.11%	89.48%
Files accessed	73.68%	78.95%	77.63%	75.00%	76.32%
Mutexes	55.26%	59.21%	59.21%	55.26%	57.24%
Registry Keys	88.16%	88.16%	86.84%	89.47%	88.16%
Average	87.89%	93.42%	92.37%	93.68%	

7.4.1 Call Traces

Although we aimed to challenge classifiers by collecting all instances from the same malware category (ransomware), whose samples should perform similar behaviours (e.g., files encryption), all the feature representations of call traces yielded promising results with an accuracy ranging from 88.16% to 96.05%. Despite the small variations caused by the different classifiers as seen in Figure 7.4, we can conclude that any feature model using call traces can be used as features to distinguish malware families (RQ1).

Considering the difference between the use of API and system calls, we can comment that models using API calls perform better on average. But the system calls have scalability advantages with a fewer number of features.

7.4.1.1 Bag-of-words Model

The simplest model extracted from call traces is the bag-of-words model without function arguments. Only 210 API functions and 35 system call functions are found. Despite the information loss due to lack of arguments and the order

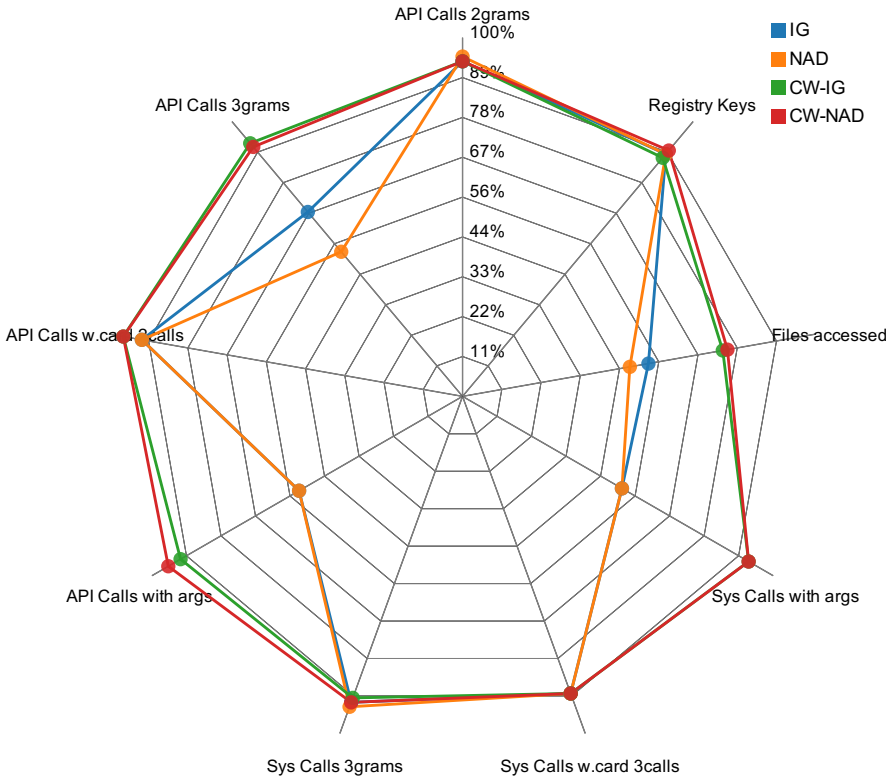


Figure 7.5: Weighted TPRs of feature models for different feature selection techniques (Neural Networks used).

between calls, the model run on API calls has yielded 93.42% accuracy, followed by system calls with 92.11%.

7.4.1.2 With Arguments

On the other hand, call traces with function arguments generated the highest number of features. As seen in Table 7.2, these models gave unique 1.9M API and 958K system calls, which make them infeasible to be adapted as ngrams (API Calls=$1.9x10^{6n}$ and System calls=$9.58x10^{5n}$). Regarding accuracy, API calls (94.74%) have performed slightly better than system calls (92.11%) with Neural Networks and Classwise NAD settings. One important benefit of using function arguments is having more insight into the called functions, whereas the function name without arguments does not reveal as much information about the intention.

Another point is that these models provide a good demonstration of how feature selection techniques can easily suffer from unfair representation without classwise

Table 7.4: Accuracy results of different feature selection methods used.

Features	IG	NAD	CWIG	CWNAD
API Calls 2grams	93.42%	94.74%	93.42%	93.42%
API Calls 3grams	67.11%	52.63%	92.11%	90.79%
API Calls w.cards 2calls	90.79%	90.79%	96.05%	96.05%
API Calls with args	52.63%	52.63%	90.79%	94.74%
Sys Calls 3grams	90.79%	92.11%	89.47%	90.79%
Sys Calls w.card 3calls	88.16%	88.16%	88.16%	88.16%
Sys Calls with args	51.32%	51.32%	92.11%	92.11%
Files accessed	52.63%	47.37%	73.68%	75.00%
Registry Keys	88.16%	88.16%	86.84%	89.47%
Average	78.13%	76.65%	90.96%	91.94%

adaptations. Since the features extracted from these models have unbalanced distributions (see Figure 7.7), naive versions of *Information Gain* and *Normalised Angular Distance* inevitably favour one class for the selected feature sets. Figure 7.8 illustrates that how the selection can differ for naive and classwise techniques on the basis of *Crysis* family. Moreover, Figure 7.5 and Table 7.4 show how the selection bias of naive techniques causes classifiers to underperform compared to the classwise selections.

7.4.1.3 Ngrams

Another feature model used is the ngram representation of call traces. Based on the average of different classifier results, 2-grams (93.75%) and 3-grams (92.11%) of system calls produce better results than the bag-of-words model (88.49%). 2-grams (92.76%) of API calls also perform better than the corresponding bag-of-words model (91.45%) as well, while 3-grams (88.49%) perform the worst among these three models. Because of the limited number of sequences that can be found on traces, ngram models are more scalable than feature models with arguments or wildcard models for which we generate all possible permutations.

7.4.1.4 Wildcard Searches

Our wildcard models are designed to be resistant to evading mechanisms such as the insertion of junk API calls. The wildcard model running on API calls for 2-calls

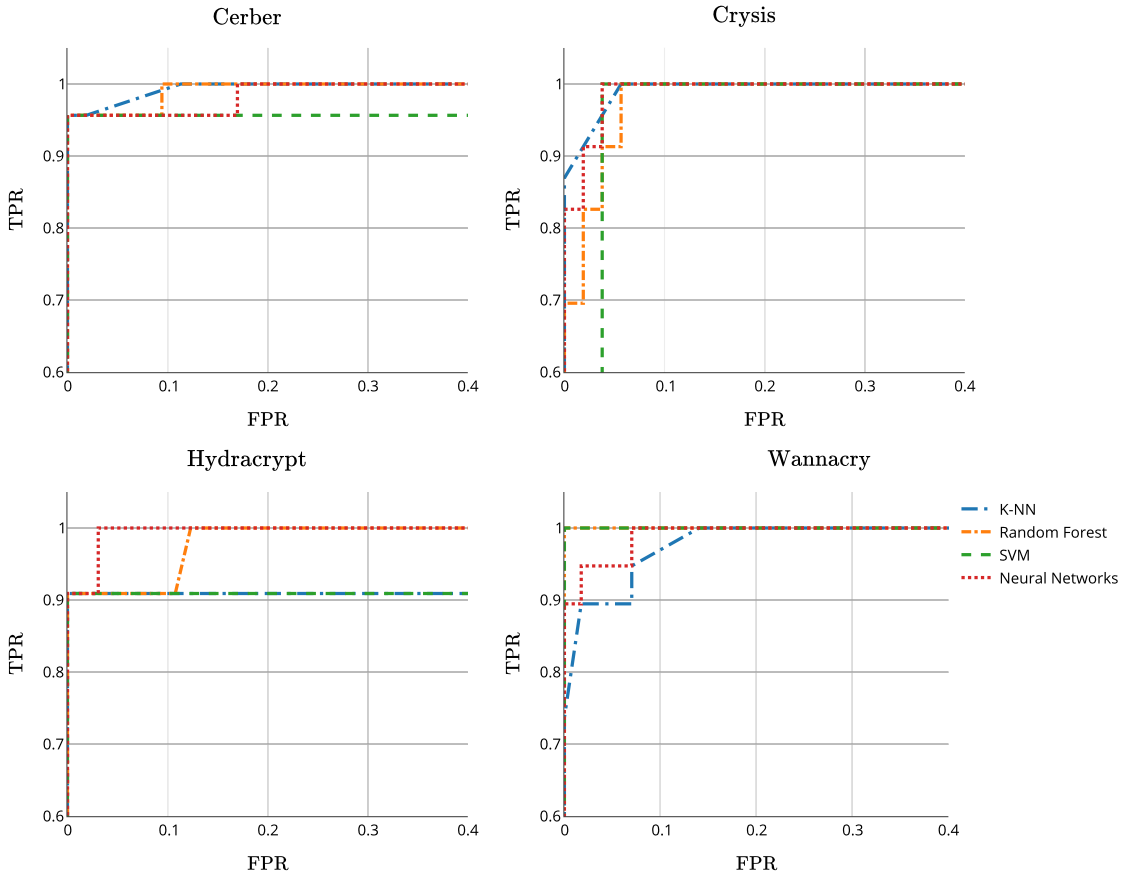


Figure 7.6: ROC curves of different classifiers for *API Calls wildcard model with 2-calls*

with a distance buffer of 4-calls yielded the best results of all experiments with an accuracy of 96.05% for Neural Networks, SVM and Random Forest, of which the ROC curves can be seen in Figure 7.6. When we compare the results with 2-grams of API calls, we can assume that some samples might be inserting junk calls and might be caught by our wildcard model.

On the system calls side, wildcard model performed slightly worse than their ngrams correspondents (see Table 7.3). The first plausible explanation can be that, due to the small feature space, wildcards on system call traces cause false-positive findings that actually do not represent any malicious behaviours. Second, as shown in the short trace examples (1), (2), due to the elimination of non-system calls from the feature space, the distance buffer that we set corresponds to the wider distances, which can be another reason for false positives.

Regarding scalability, we normally used hash tables to store all feature models.

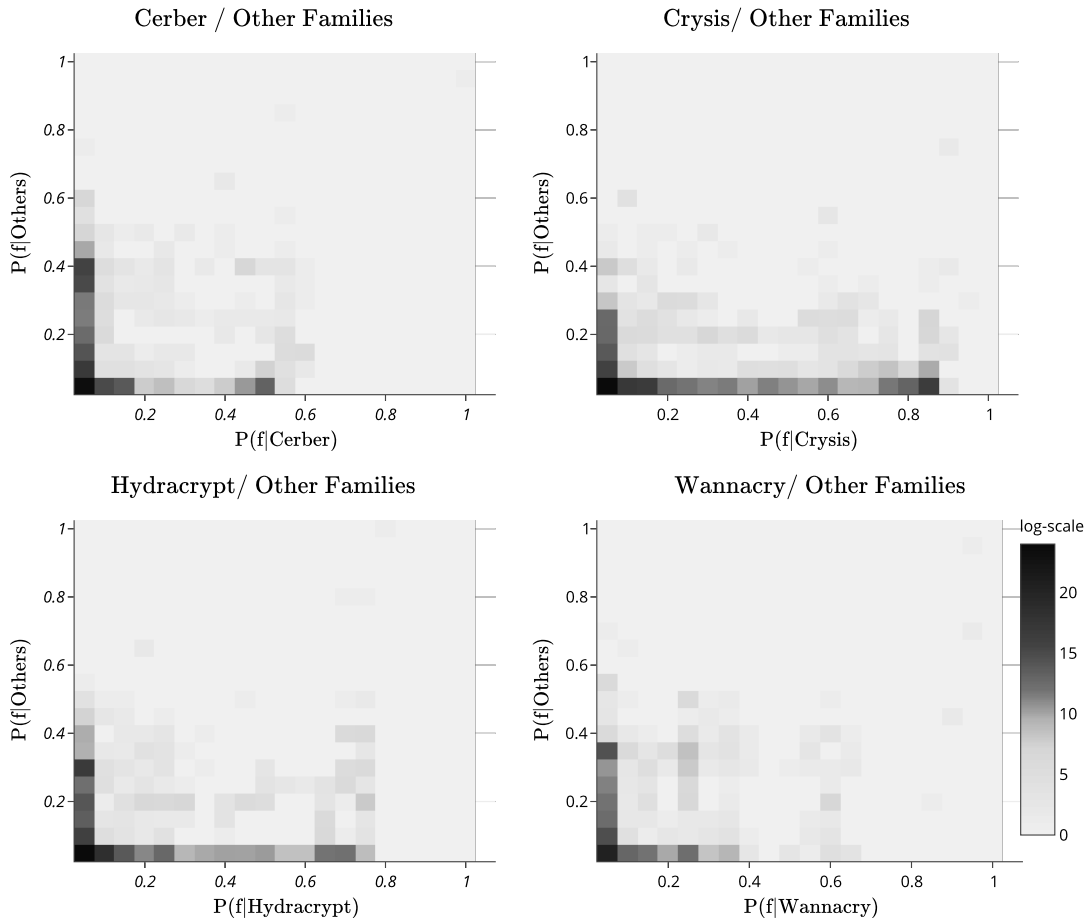


Figure 7.7: Family-wise distributions of the features extracted from *API Calls with args* (1.9M features extracted).

This enabled us to count and analyse all the features for the given trace (N) with a $O(N)$ complexity in total. However, in order to analyse the existence or absence of a wildcard feature on a given call trace, we ran regular expressions for each feature that brings an additional $O(N)$ complexity multiplied by the number of wildcard permutations for one sample trace. Moreover, since we had to generate all the possible permutations as wildcard features in advance, $P(n, r)$ becomes infeasible with $r > 3$ and n for a cardinality of hundreds where the Windows has more than 300 system calls, though we only found 35 in our dataset. For RQ2, we can conclude that the wildcard model for API calls with 2-calls yields the best accuracy of the experiments, while 2-grams and 3-grams perform better than the other models extracted from system calls. Although there are signs of resilience against junk API calls by wildcard models, those that run on system calls suffer

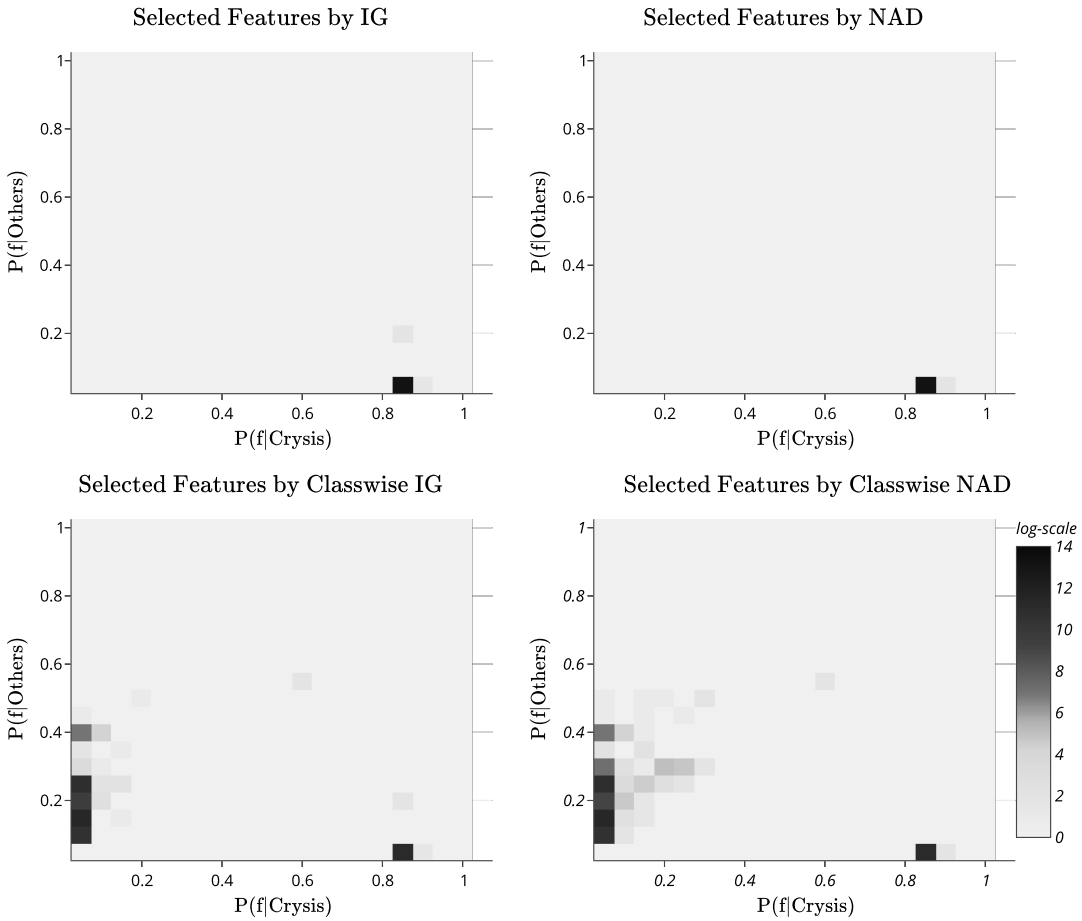


Figure 7.8: *Crysis*-based distributions of selected features by different selection techniques for *API Calls with args* (10K features selected).

from false positives, possibly due to the small size of the feature space.

7.4.2 Other Artefacts

In addition to the models extracted from call traces, other feature models based on *registry keys* and *DLLs loaded* also produced promising results whereas the *files accessed* and *mutexes* performed poorly than expected.

DLLs and registry keys used during the analyses yielded 89.45% and 88.16% accuracy on average, respectively. The names of the files accessed produced 76.32% correct classification ratio, while mutex names was the worst performing model with a 57.24% (RQ3).

Table 7.5: Significance test results ($\alpha = 0.05$) to evaluate feature selection methods and classifiers

Selection	NN=RF	Classifier	IG=NAD	Naive=CW
IG	reject	KNN	do not reject	reject
NAD	reject	RF	do not reject	reject
CW-IG	do not reject	SVM	do not reject	reject
CW-NAD	do not reject	NN	do not reject	reject

7.4.3 Optimal Settings and Comparison

As a response to our RQ4, in terms of classifier algorithms, Neural Networks and Random Forest performed quite well with classwise selection techniques, though Neural Networks performed worse than Random Forest for naive ones (see Table 7.3 and 7.5).

On the other hand, our experiment results highlight that unfair representation of classes by the selected feature sets is an important issue that needs to be addressed by selection techniques. Although naive selection methods suffer from poor classification results for large feature spaces without any significant difference between naive *Information Gain* and *Normalised Angular Distance*, our classwise adaptations produce significantly more accurate results (see Table 7.4 and 7.5).

With Neural Networks and Classwise NAD settings, the wildcard model of API calls for 2-calls became the best performing feature model of our experiments, whose detailed performance metrics can be seen in Table 7.6. Although our study had to use a dataset consists of fewer samples for a sound approach that is free category based biases, it outperformed most similar studies performing family classification [146] and categorisation [145].

7.4.4 Limitations

Despite the focus of the chapter on the performance comparison of different feature models and selection techniques, there are some factors defining the success over which we did not have much control. For example, even though the queries made by family names returned too many files, filtering them with a labelling tool [182]

Table 7.6: Class-based performance metrics and comparison with related work.

	TPR	**FPR**	**Prec.**	**Recall**	**F-M**	**ROC**
Hansen et al [146]	0.864	0.035	0.872	N/A	0.864	0.978
Pirscoveanu et al [145]	0.896	0.049	0.907	N/A	0.898	0.980
cerber	0.957	0.019	0.957	0.957	0.957	0.993
crysis	1	0.038	0.92	1	0.958	0.995
hydracrypt	0.909	0	1	0.909	0.952	0.996
wannacry	0.947	0	1	0.947	0.973	0.999
Weighted	0.961	0.017	0.963	0.961	0.961	0.995

reduced the number of usable samples significantly. Because we applied supervised-learning and our classifiers used the most popular AV labels for training, the performance of the classifiers was dependent on the way how these AV engines analyse and label these samples.

The work here also relied on analysis reports generated by Cuckoo sandbox. Although Cuckoo integrates some mechanisms to avoid the detection of the analysis environment by samples, there may be samples hiding their malicious behaviour and acting like legitimate software within the sandbox.

7.5 Summary

This chapter presented an ML-based malware analysis framework to explore the use of system-level runtime features for the identification of malware families. The framework mainly looked at different feature types extracted from API call traces, representing malware's interactions with the host operating system. Although all the feature models extracted from these yielded promising results, the wildcard patterns outperformed the other models. Furthermore, the chapter investigated different feature selection methods, i.e., a new method that can outweigh the distinctiveness of a feature over its popularity, and their classwise adaptations to address biases caused by imbalanced feature distributions.

"Science is the great antidote to the poison of enthusiasm and superstition."

— Adam Smith

8 Conclusion

This thesis offered multiple runtime protections for software programs, the execution of which can be compromised via memory corruptions. The focus was on sophisticated runtime attacks that carefully craft control and (non-control) data variables, instead of simpler attack scenarios that alter the original code or inject new code. During the design of these protections, we deliberately avoided asking for intrusive hardware changes, such as ISA modifications, which would not fit the majority of existing and legacy systems. Instead, we have aimed for both practical and strong security that can be bought externally or adapted through only software changes.

8.1 Summary of Contributions

This thesis started by providing background information for readers who may not be familiar with the potential threats to software runtime and attestation protocols. Also, a comprehensive review of relevant memory protection and attestation literature was given in Chapter 3, which can be used to confirm the research gaps and challenges we have identified.

To address these gaps, Chapter 4 first proposed *a conceptual hardware module (HSM) that can act like a trust anchor for runtime attestation* on critical embedded systems. Unlike previous attestation schemes that record execution traces or digest

them into a hash value, our approach suggests performing the necessary checks in real time without accumulating any trace information. Therefore, it avoids not only storage and communication overheads but also asking the verifier to discover all acceptable hash values (e.g., CFG paths), so any potential state or path explosion issues. To reason about the correctness of runtime, *our approach relies on a lightweight static model (RIM)* of the software subject to attestation, i.e., a call-graph like model loaded into the memory of hardware module. For better precision on return address checks, the scheme also employs call counters to handle unbounded stack layouts that recursive functions would generate. With an external hardware component that only needs bus integration, the chapter aims to offer an attestation solution that can fit to existing systems.

Second, Chapter 5 introduced *a novel and automated way to identify critical program variables* whose modification can result in a meaningful data-oriented attack. The proposed distinction is based on the trustworthiness of agents contributing variable values, as we consider the variables defined by trusted agents (e.g., programmer) must be primarily protected compared to variables that are already controllable by untrusted agents (e.g., users). The chapter described a static analysis method that identifies the propagation of trust among program variables by examining their control and data dependencies. This method not only helps to implement *a lightweight targeted protection scheme against data attacks* but also provides guidance for any isolation mechanism that lacks knowledge of what to isolate without the help of the programmer.

Chapter 6 offered a *compiler-based scheme that systematically utilises CPU registers to provide strong integrity protection on critical variables* in use. In contrast to conventional performance-oriented register allocations that favour variables with higher use densities, the chapter first described a security-oriented allocation strategy that favours variables that are more likely to be targeted for a successful attack. This approach can be especially meaningful for CPU architectures with register scarcity to reduce the attack surface in the best possible way. Second, the chapter leveraged cryptographic primitives to ensure the integrity of register data (spills)

at rest across function calls. This assurance enables us to repeatedly use the register file as secure storage for each function call, which is typically enough to accommodate all the variables of a single function.

Lastly, Chapter 7 presented *a machine-learning based malware analysis framework for the evaluation of system-level runtime features* that can be used to identify malicious software executions. This framework does not offer a solution against exploit-based attacks, unlike previous chapters. Instead, it targets a system setting that is already infected with a malicious program whose static analysis might not be an option due to sophisticated evading mechanisms (e.g., polymorphism). The framework uses call traces collected at the system level and explores how their different representations can enhance classifier results.

8.2 Concluding Remarks and Future Outlook

This thesis represents an important step towards obtaining stronger security guarantees from devices that are vulnerable to software attacks. The thesis made it possible for a remote party to ask for integrity assurance beyond the load-time software correctness. A control-oriented (e.g., code-reuse) attack scenario, which would normally go unnoticed by a static attestation method, can be revealed using the methods proposed in this thesis. Furthermore, it demonstrated how to extend current runtime protections usable in practice (i.e., control-flow), in a way that they can also address data-oriented attacks without causing a big leap in performance overheads. Lastly, in this thesis, we enabled a simple computing device, which lacks any special hardware security feature, to minimise its memory attack surface against both control and data attacks, using a very fundamental building block of any computer architecture, i.e., CPU registers. In addition to these practical consequences, the following sections share insights and lessons on how the community can adopt more practical solutions to these problems in the future.

Runtime Attestation. Two separate branches have emerged in the literature to address runtime integrity problems: The first group of mitigation methods (e.g., CFI [2]) typically validates runtime correctness according to a static model (e.g., CFG) in real time and terminates execution in the event of a violation. On the contrary, the second group of runtime attestation schemes (e.g., CFA [5, 6]) defers this task and offloads it to the verifier by sending information about past runtime states (traces) without terminating the execution. Although both parties have substantial overlap considering the formulation of attacks, attestation studies pose many challenges that prevent them from being adopted in practice. For instance, the schemes digesting runtime traces [5, 6] require the verifier to discover all valid measurements, using the reference model (e.g., CFG). But this discovery process may not halt even for a moderate program due to the combinatorial explosion in the number of acceptable paths by the static model. For the schemes [7, 8] reporting traces to the verifier in a lossless way, the main challenge becomes network overheads and other difficulties of storing large amount of traces on the prover's memory. Lastly, some attestation proposals [6, 15] also consider that the program input is provided by the verifier. Unfortunately, this is an unrealistic assumption as the program input is typically given by the prover's context. Otherwise, as a trusted party, the verifier could perform the computation task itself without any need for a prover device.

Therefore, we believe that prospective attestation studies should also aim for runtime checks on the prover side in real-time, using appropriate trust anchors and static program models. Such an approach would help to formulate remote runtime attestation as more like a reporting task without program termination. We remind the reader that these static graph models, which can serve as a FSM, already correspond to the set of acceptable runtime traces, (i.e., the corresponding regular language). Hence, with a real-time approach, there would not be any need for accumulation or transfer of the trace information to the verifier. Otherwise, in a setting where program termination is permissible, static attestation (e.g., measured boot) of the program code equipped with attack mitigation techniques such as CFI would suffice to establish trust in a system.

Practical Mitigation of Data-Oriented Attacks. In order to address control-oriented attacks, the literature provides many practical solutions such as CFI [2] and CPS [3]. Thanks to their targeted approach, focusing only on control variables (i.e., code pointers), these schemes enjoy lightweight performance costs and better data separability at compile time, making them usable in practice. However, control variables constitute only a minor portion compared to the rest of the program data, whose complete protection hinders the practicality and adaptability of proposed schemes, such as DFI [4].

Therefore, we assume that there is research value to work towards practical mitigation of data-orientated attacks. Chapter 5 and recent papers such as DataShield [14], CVI (OAT) [71] can be considered as examples of such efforts that avoid the inspection of every memory access, by focusing on critical (non-control) data given at compile time. These solutions can also be used by in-process isolation primitives [9, 12] lacking an automated method to identify the data in need of isolation for integrity purposes. As an orthogonal but relevant problem, we remind the reader that the efforts on improving pointer analysis techniques can also directly contribute to the quality of those schemes by providing better data separability. A pointer analysis that is both scalable and precise (i.e., flow-sensitive) would enhance any runtime protection relying on a static approximation. Otherwise, dynamic methods such as DIFT [123] are required for precision, which generally requires disruptive changes in the architecture and in the data paths.

Memory Safety. The alternative way to avoid both control- and data-oriented attacks is memory safety that can eliminate bugs and their exploits as the root cause. However, memory safety solutions implemented via software-based runtime mechanisms (e.g., bound-checking) are generally costly to be adopted in practice. In this regard, as a safe language by design, Rust [191] is a strong candidate that can replace C-like system-programming languages in the long term, without sacrificing the performance, thanks to its safety properties enforced primarily at compile-time. But the impracticality of rewriting existing software stacks in a new language and

the steep learning curve awaiting Rust developers might slow down the adaptation of this language. Therefore, to solve memory safety issues problems of existing vulnerable software stacks with small touches on the code, CHERI [91] as a capability architecture represents a hardware-based solution that is likely to be adopted in practice in the future, with strong industrial interest and support. CHERI suggests changing integer-like language pointers with architectural fat pointers (capabilities) where each dereference is audited based on the bounds information available. Thanks to being a ISA-based hardware solution, CHERI makes bound checking possible with reasonable runtime overheads. We can expect both Rust and CHERI to be the game changers that can finally solve long-standing memory safety problems.

Hardware and Legacy Systems. In general, hardware-based protections can address more powerful adversaries with better performance, whereas software-based methods are subject to higher overhead costs and are easier to bypass. Despite their benefits, the biggest issue with hardware-based approaches is deployability challenges. Even if these new designs are put into production by manufacturers and trigger changes in future architectures, existing devices will remain vulnerable, and are unlikely to be replaced for years, maybe decades. Considering the billions of embedded devices already out there, such as sensor systems and IoT, the security of existing devices is a critical problem that the research community should invest in more. Hence, we believe future work should seek ways to protect existing systems through software-only solutions or non-invasive solutions where the security can be externally provided. Chapters 4 and 6 can be seen as positive steps towards this goal.

Cryptographic Approaches. The correctness of a runtime protection method is typically dependent on the integrity of the instrumentation or verification data (e.g., shadow stack) generated. Due to practical reasons such as switch costs, those data need to be located within the same address space, which requires practical in-process isolation techniques. Previous software- (e.g., SFI [11]) and hardware-based (e.g., HDFI [9]) isolations have different strengths and weaknesses. Alternative to those, cryptographic methods, such as encryption to mask critical data or MAC

to check their integrity, can avoid generation of additional instrumentation data and can eliminate the need for an isolation mechanism. Hence, the problem can be reduced to the confidentiality of the keys that CPU registers can easily secure as we suggested in Chapter 6. Thanks to recent ISA advancements that can significantly accelerate cryptographic operations within the CPU (e.g., PAC), we can anticipate more studies in this direction to protect runtime.

References

[1] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without Function Calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. New York, New York, USA: ACM Press, 2007, p. 552. DOI: 10.1145/1315245.1315313.

[2] Martín Abadi et al. "Control-flow integrity". In: *Proceedings of the 12th ACM conference on Computer and communications security - CCS '05*. New York, New York, USA: ACM Press, 2005, p. 340. DOI: 10.1145/1102120.1102165.

[3] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. "Code-pointer integrity". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. October. 2014, pp. 147–163.

[4] Castro Miguel, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity". In: *Proceedings of the 7th symposium on Operating systems design and implementation - USENIX Association*. USENIX Association, 2006, pp. 147–160.

[5] Tigist Abera et al. "C-FLAT: Control-Flow Attestation for Embedded Systems Software". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2016, pp. 743–754. DOI: 10.1145/2976749.2978358.

[6] Ghada Dessouky et al. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In: *Proceedings of the 54th Annual Design Automation Conference 2017*. New York, NY, USA: ACM, June 2017, pp. 1–6. DOI: 10.1145/3061639.3062276.

[7] Ghada Dessouky et al. "LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution". In: *Proceedings of the International Conference on Computer-Aided Design*. New York, NY, USA: ACM, Nov. 2018, pp. 1–8. DOI: 10.1145/3240765.3240821.

[8] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. "Tiny-CFA: Minimalistic Control-Flow Attestation Using Verified Proofs of Execution". In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Vol. 2021-Febru. IEEE, Feb. 2021, pp. 641–646. DOI: 10.23919/DATE51398.2021.9474029.

[9] Chengyu Song et al. "HDFI: Hardware-Assisted Data-Flow Isolation". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2016, pp. 1–17. DOI: 10.1109/SP.2016.9.

[10] Enes Göktaş et al. *Bypassing clang's SafeStack for Fun and Profit*. 2016. URL: https://www.blackhat.com/docs/eu-16/materials/eu-16-Goktas-Bypassing-Clangs-SafeStack.pdf.

[11] Stephen McCamant and Greg Morrisett. “Evaluating SFI for a CISC architecture”. In: *15th USENIX Security Symposium*. 2006, pp. 209–224.

[12] Tommaso Frassetto et al. “IMIX: Hardware-Enforced In-Process Memory Isolation”. In: *Proceedings of the 27th USENIX Security Symposium - USENIX Security '18*. 2018, pp. 83–97.

[13] Chengyu Song et al. “Enforcing Kernel Security Invariants with Data Flow Integrity”. In: *Proceedings 2016 Network and Distributed System Security Symposium*. February. Reston, VA: Internet Society, 2016, pp. 21–24. DOI: 10.14722/ndss.2016.23218.

[14] Scott A. Carr and Mathias Payer. “DataShield: Configurable data confidentiality and integrity”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. New York, NY, USA: ACM, Apr. 2017, pp. 193–204. DOI: 10.1145/3052973.3052983.

[15] Boyu Kuang et al. “DO-RA: Data-oriented runtime attestation for IoT devices”. In: *Computers & Security* 97 (Oct. 2020), p. 101945. DOI: 10.1016/j.cose.2020.101945.

[16] Frederik Armknecht et al. “A security framework for the analysis and design of software attestation”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. New York, New York, USA: ACM Press, 2013, pp. 1–12. DOI: 10.1145/2508859.2516650.

[17] Rodrigo Vieira Steiner and Emil Lupu. “Attestation in Wireless Sensor Networks”. In: *ACM Computing Surveys* 49.3 (Dec. 2016), pp. 1–31. DOI: 10.1145/2988546.

[18] Glenn Wurster, P.C. van Oorschot, and Anil Somayaji. “A Generic Attack on Checksumming-Based Software Tamper Resistance”. In: *2005 IEEE Symposium on Security and Privacy (S&P'05)*. IEEE, 2005, pp. 127–138. DOI: 10.1109/SP.2005.2.

[19] Arvind Seshadri et al. “Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems”. In: *Proceedings of the twentieth ACM symposium on Operating systems principles - SOSP '05*. Vol. 39. 5. New York, New York, USA: ACM Press, 2005, p. 1. DOI: 10.1145/1095810.1095812.

[20] Claude Castelluccia et al. “On the difficulty of software-based attestation of embedded devices”. In: *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*. New York, New York, USA: ACM Press, 2009, p. 400. DOI: 10.1145/1653662.1653711.

[21] László Szekeres et al. “SoK: Eternal War in Memory”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE, May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[22] Crispan Cowan et al. “StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks”. In: *Proceedings of the 7th USENIX Security Symposium*. San Antonio, TX: USENIX Association, 1998, pp. 63–78.

[23] Alexander Peslyak. *Bugtraq: Getting around non-executable stack (and fix)*. 1997. URL: http://seclists.org/bugtraq/1997/Aug/63.

[24] Sebastian Krahmer. *X86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique*. 2005. URL: http://users.suse.com/~krahmer/no-nx.pdf.

[25] PaX Team. *PaX address space layout randomization (ASLR)*. 2003. URL: https://pax.grsecurity.net/docs/aslr.txt.

[26] Tyler Bletsch et al. "Jump-oriented programming: A New Class of Code-Reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. New York, New York, USA: ACM Press, 2011, p. 30. DOI: 10.1145/1966913.1966919.

[27] Erik Buchanan et al. "When good instructions go bad: Generalizing Return-Oriented Programming to RISC". In: *Proceedings of the 15th ACM conference on Computer and communications security - CCS '08*. New York, New York, USA: ACM Press, 2008, p. 27. DOI: 10.1145/1455770.1455776.

[28] Stephen Checkoway et al. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*. New York, New York, USA: ACM Press, 2010, p. 559. DOI: 10.1145/1866307.1866370.

[29] Shuo Chen et al. "Non-control-data attacks are realistic threats". In: *USENIX Security Symposium*. Vol. 5. 2005.

[30] Nicholas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". In: *USENIX Security Symposium*. 2015, pp. 161–176.

[31] Hong Hu et al. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[32] Kyriakos K. Ispoglou et al. "Block Oriented Programming: Automating Data-Only Attacks". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2018, pp. 1868–1882. DOI: 10.1145/3243734.3243739.

[33] Periklis Akritidis et al. "Preventing Memory Error Exploits with WIT". In: *2008 IEEE Symposium on Security and Privacy (sp 2008)*. IEEE, May 2008, pp. 263–277. DOI: 10.1109/SP.2008.30.

[34] Isaac Evans et al. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity". In: *2015 IEEE Symposium on Security and Privacy*. IEEE, May 2015, pp. 781–796. DOI: 10.1109/SP.2015.53.

[35] Gregory J. Chaitin et al. "Register allocation via coloring". In: *Computer Languages* 6.1 (1981), pp. 47–57. DOI: 10.1016/0096-0551(81)90048-5.

[36] Keith D. Cooper and L. Taylor Simpson. "Live range splitting in a graph coloring register allocator". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 1383. 1998, pp. 174–187. DOI: 10.1007/BFb0026430.

[37] Christian Wimmer and Hanspeter Mössenböck. "Optimized interval splitting in a linear scan register allocator". In: *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments - VEE '05*. New York, New York, USA: ACM Press, 2005, p. 132. DOI: 10.1145/1064979.1064998.

[38] G. J. Chaitin. "Register allocation and spilling via graph coloring". In: *ACM SIGPLAN Notices* 17.6 (June 1982), pp. 98–101. DOI: 10.1145/872726.806984.

[39] L. P. Horwitz et al. “Index Register Allocation”. In: *Journal of the Association for Computing Machinery* 13.1 (1966), pp. 43–61.

[40] Frederick Chow and John Hcnnessy. “Register allocation by priority-based coloring”. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, SIGPLAN 1984* 19.6 (1984), pp. 222–232. DOI: 10.1145/502874.502896.

[41] Vatsa Santhanam and Daryl Odnert. “Register allocation across procedure and module boundaries”. In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation - PLDI '90*. Vol. 25. 6. New York, New York, USA: ACM Press, 1990, pp. 28–39. DOI: 10.1145/93542.93551.

[42] Fred C. Chow and John L. Hennessy. “The priority-based coloring approach to register allocation”. In: *ACM Transactions on Programming Languages and Systems* 12.4 (Oct. 1990), pp. 501–536. DOI: 10.1145/88616.88621.

[43] Preston Briggs, Keith D. Cooper, and Linda Torczon. “Improvements to graph coloring register allocation”. In: *ACM Transactions on Programming Languages and Systems* 16.3 (May 1994), pp. 428–455. DOI: 10.1145/177492.177575.

[44] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation”. In: *ACM Transactions on Programming Languages and Systems* 21.5 (Sept. 1999), pp. 895–913. DOI: 10.1145/330249.330250.

[45] Christian Wimmer and Michael Franz. “Linear scan register allocation on SSA form”. In: *Proceedings of the 8th annual IEEE/ ACM international symposium on Code generation and optimization - CGO '10*. New York, New York, USA: ACM Press, 2010, p. 170. DOI: 10.1145/1772954.1772979.

[46] Omri Traub, Glenn Holloway, and Michael D. Smith. “Quality and Speed in Linear-scan Register Allocation”. In: *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)* 33.5 (1998), pp. 142–151.

[47] Diomidis Spinellis. “Reflection as a mechanism for software integrity verification”. In: *ACM Transactions on Information and System Security* 3.1 (Feb. 2000), pp. 51–62. DOI: 10.1145/353323.353383.

[48] Arvind Seshadri et al. “SWATT: software-based attestation for embedded devices”. In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.* IEEE, 2004, pp. 272–282. DOI: 10.1109/SECPRI.2004.1301329.

[49] Tamer AbuHmed, Nandinbold Nyamaa, and DaeHun Nyang. “Software-Based Remote Code Attestation in Wireless Sensor Network”. In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. IEEE, Nov. 2009, pp. 1–8. DOI: 10.1109/GLOCOM.2009.5425280.

[50] Yi Yang et al. “Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks”. In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, Oct. 2007, pp. 219–230. DOI: 10.1109/SRDS.2007.4365698.

[51] Reiner Sailer et al. “Design and Implementation of a TCG-based Integrity Measurement Architecture”. In: *USENIX Security Symposium*. Vol. 13. 2004, pp. 223–238. DOI: 10.1109/MSP.2010.92.

[52] *TPM Library Specification | Trusted Computing Group*. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[53] Trusted Computing Group. *TCPA Specification Version 1.1b*. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCPA_Main_TCG_Architecture_v1_1b.pdf.

[54] Trusted Computing Group. *TCG TPM Main Part 1 Design Principles Version 1.2*. URL: https://trustedcomputinggroup.org/wp-content/uploads/tpmwg-mainrev62_Part1_Design_Principles.pdf.

[55] Intel Corporation. *Intel® Software Guard Extensions - Developer Guide*. 2016. URL: https://download.01.org/intel-sgx/linux-1.7/docs/Intel_SGX_Developer_Guide.pdf.

[56] Jonathan M. McCune et al. "Flicker: An Execution Infrastructure for TCB Minimization". In: *ACM SIGOPS Operating Systems Review* 42.4 (Apr. 2008), pp. 315–328. DOI: 10.1145/1357010.1352625.

[57] Intel Corporation. *Intel® Trusted Execution Technology - Software Development Guide*. 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf.

[58] ARM. *ARM Security Technology: Building a Secure System using TrustZone Technology*. 2009. URL: https://bit.ly/1g6dOxn.

[59] Joonho Kong et al. "PUFatt: Embedded Platform Attestation Based on Novel Processor-Based PUFs". In: *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*. New York, New York, USA: ACM Press, 2014, pp. 1–6. DOI: 10.1145/2593069.2593192.

[60] Karim El Defrawy et al. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." In: *Ndss*. Vol. 12. 2012, pp. 1–15.

[61] Patrick Koeberl et al. "TrustLite: A Security Architecture for Tiny Embedded Devices". In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*. New York, New York, USA: ACM Press, 2014, pp. 1–14. DOI: 10.1145/2592798.2592824.

[62] Ferdinand Brasser et al. "TyTAN: Tiny trust anchor for tiny devices". In: *Proceedings of the 52nd Annual Design Automation Conference*. New York, NY, USA: ACM, June 2015, pp. 1–6. DOI: 10.1145/2744769.2744922.

[63] Aurelien Francillon et al. "A minimalist approach to Remote Attestation". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014*. New Jersey: IEEE Conference Publications, 2014, pp. 1–6. DOI: 10.7873/DATE.2014.257.

[64] Aurelien Francillon et al. "Systematic Treatment of Remote Attestation". In: *IACR Cryptology ePrint Archive*. 2012, p. 713.

[65] Ivan De Oliveira Nunes et al. "VRased: A verified hardware/software co-design for remote attestation". In: *Proceedings of the 28th USENIX Security Symposium*. 2019, pp. 1429–1446.

[66] Ivan de Oliveira Nunes et al. “PURE: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in low-End Embedded Systems”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Vol. 2019-Novem. IEEE, Nov. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942118.

[67] Ivan De Oliveira Nunes et al. “APEX: A verified architecture for proofs of execution on remote devices under full software compromise”. In: *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 2020, pp. 771–788.

[68] Ivan De Oliveira Nunes et al. “On the TOCTOU Problem in Remote Attestation”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Nov. 2021, pp. 2921–2936. DOI: 10.1145/3460120.3484532.

[69] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. “Dynamic integrity measurement and attestation: Towards Defense Against Return-Oriented Programming Attacks”. In: *Proceedings of the 2009 ACM workshop on Scalable trusted computing - STC ’09*. New York, New York, USA: ACM Press, 2009, p. 49. DOI: 10.1145/1655108.1655117.

[70] Chongkyung Kil et al. “Remote attestation to dynamic system properties: Towards providing complete system integrity evidence”. In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, June 2009, pp. 115–124. DOI: 10.1109/DSN.2009.5270348.

[71] Zhichuang Sun et al. “OAT: Attesting Operation Integrity of Embedded Devices”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. Vol. 2020-May. IEEE, May 2020, pp. 1433–1449. DOI: 10.1109/SP40000.2020.00042.

[72] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. “DIALED: Data Integrity Attestation for Low-end Embedded Devices”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. Vol. 2021-Decem. IEEE, Dec. 2021, pp. 313–318. DOI: 10.1109/DAC18074.2021.9586180.

[73] Xinyu Jin et al. “Unpredictable Software-based Attestation Solution for node compromise detection in mobile WSN”. In: *2010 IEEE Globecom Workshops*. IEEE, Dec. 2010, pp. 2059–2064. DOI: 10.1109/GLOCOMW.2010.5700307.

[74] Ernie Brickell, Jan Camenisch, and Liqun Chen. “Direct anonymous attestation”. In: *Proceedings of the 11th ACM conference on Computer and communications security - CCS ’04*. New York, New York, USA: ACM Press, 2004, p. 132. DOI: 10.1145/1030083.1030103.

[75] N. Asokan et al. “SEDA: Scalable Embedded Device Attestation”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2015, pp. 964–975. DOI: 10.1145/2810103.2813670.

[76] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. “SALAD: Secure and lightweight attestation of highly dynamic and disruptive networks”. In: *ASIACCS 2018 - Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security* (2018), pp. 329–342. DOI: 10.1145/3196494.3196544.

[77] Xavier Carpent et al. “Lightweight swarm attestation: A tale of two LISA-s”. In: *ASIA CCS 2017 - Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security* (2017), pp. 86–100. DOI: 10.1145/3052973.3053010.

[78] Ahmad Ibrahim, Ahmad Reza Sadeghi, and 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 2017* (2017), pp. 64–74. DOI: 10.1145/3098243.3098260.

[79] Nicholas Nethercote and Julian Seward. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In: *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’07*. New York, New York, USA: ACM Press, 2007, p. 89. DOI: 10.1145/1250734.1250746.

[80] Reed Hastings. “Purify: Fast detection of memory leaks and access errors”. In: *Proc. 1992 Winter USENIX Conference*. 1992, pp. 125–136.

[81] Konstantin Serebryany and Derek Bruening. “AddressSanitizer: a fast address sanity checker”. In: *USENIX Annual Technical Conference*. 2012, pp. 309–318. URL: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf.

[82] Richard W M Jones and Paul H J Kelly. “Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs.” In: *Third International Workshop on Automated Debugging*. Citeseer. 1997.

[83] Olatunji Ruwase and Monica S Lam. “A Practical Dynamic Buffer Overflow Detector”. In: *NDSS*. 2004, pp. 159–169.

[84] Periklis Akritidis et al. “Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.” In: *USENIX Security Symposium*. 2009, pp. 51–66.

[85] George C. Necula, Scott McPeak, and Westley Weimer. “CCured: Type-Safe Retrofitting of Legacy Code”. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL ’02*. New York, New York, USA: ACM Press, 2002, pp. 128–139. DOI: 10.1145/503272.503286.

[86] Dan Grossman et al. “Cyclone: A Safe Dialect of C”. In: *USENIX 2002 Annual Technical Conference*. 2002, pp. 275–288. URL: http://shootout.alioth.debian.org/..

[87] Joe Devietti et al. “HardBound: Architectural Support for Spatial Safety of the C Programming Language”. In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems - ASPLOS XIII*. Vol. 36. 1. New York, New York, USA: ACM Press, 2008, p. 103. DOI: 10.1145/1346281.1346295.

[88] Santosh Nagarakatte et al. “SoftBound: Highly Compatible and Complete Spatial Memory Safety for C”. In: *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’09*. Vol. 44. 6. New York, New York, USA: ACM Press, 2009, p. 245. DOI: 10.1145/1542476.1542504.

[89] Intel Corporation. *Support for Intel® Memory Protection Extensions (Intel® MPX)...* URL: https://www.intel.com/content/www/us/en/support/articles/000059823/processors.html.

[90] Albert Kwon et al. "Low-fat pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-based Security". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. New York, New York, USA: ACM Press, 2013, pp. 721–732. DOI: 10.1145/2508859.2516713.

[91] Jonathan Woodruff et al. "The CHERI capability model: Revisiting RISC in an age of risk". In: *Proceedings - International Symposium on Computer Architecture* (2014), pp. 457–468. DOI: 10.1109/ISCA.2014.6853201.

[92] Byoungyoung Lee et al. "Preventing Use-after-free with Dangling Pointers Nullification". In: *Proceedings 2015 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, Feb. 2015. DOI: 10.14722/ndss.2015.23238.

[93] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. "DangSan: Scalable Use-after-free Detection". In: *Proceedings of the Twelfth European Conference on Computer Systems*. New York, NY, USA: ACM, Apr. 2017, pp. 405–419. DOI: 10.1145/3064176.3064211.

[94] Daiping Liu, Mingwei Zhang, and Haining Wang. "A Robust and Efficient Defense against Use-after-Free Exploits via Concurrent Pointer Sweeping". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018). DOI: 10.1145/3243734.

[95] Santosh Nagarakatte et al. "CETS: Compiler-Enforced Temporal Safety for C". In: *Proceedings of the 2010 international symposium on Memory management - ISMM '10*. Vol. 45. 8. New York, New York, USA: ACM Press, 2010, p. 31. DOI: 10.1145/1806651.1806657.

[96] *PTAuth: Temporal Memory Safety via Robust Points-to Authentication | USENIX*. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[97] Arm. *Memory Tagging Extension, Armv8.5-A*. Tech. rep., pp. 1–9.

[98] Lucas Davi et al. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones". In: *NDSS 2012 (19th Network and Distributed System Security Symposium)*. Vol. 26. 2012, pp. 27–40.

[99] Fardin Abdi Taghi Abad et al. "On-chip control flow integrity check for real time embedded systems". In: *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*. IEEE, Aug. 2013, pp. 26–31. DOI: 10.1109/CPSNA.2013.6614242.

[100] Ben Niu and Gang Tan. "Modular control-flow integrity". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, June 2014, pp. 577–587. DOI: 10.1145/2594291.2594295.

[101] Mingwei Zhang and R. Sekar. "Control Flow for COTS binaries". In: *Proceedings of the 31st Annual Computer Security Applications Conference on - ACSAC 2015*. New York, New York, USA: ACM Press, 2015, pp. 91–100. DOI: 10.1145/2818000.2818016.

[102] Ali Jose Mashtizadeh et al. "CCFI: Cryptographically Enforced Control Flow Integrity". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15* (Aug. 2014), pp. 941–951. DOI: 10.1145/2810103.2813676.

[103] Lucas Davi et al. "HAFIX: Hardware-Assisted Flow Integrity eXtension". In: *Proceedings of the 52nd Annual Design Automation Conference*. New York, NY, USA: ACM, June 2015, pp. 1–6. DOI: 10.1145/2744769.2744847.

[104] Nick Christoulakis et al. "HCFI: Hardware-enforced Control-Flow Integrity Nick". In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. New York, NY, USA: ACM, Mar. 2016, pp. 38–49. DOI: 10.1145/2857705.2857722.

[105] Mario Werner et al. "Sponge-Based Control-Flow Protection for IoT Devices". In: *arXiv preprint arXiv:1802.06691*. 2018. URL: http://arxiv.org/abs/1802.06691.

[106] Nathan Burow et al. "Control-Flow Integrity". In: *ACM Computing Surveys* 50.1 (Jan. 2018), pp. 1–33. DOI: 10.1145/3054924.

[107] Frances E. Allen. "Control Flow Analysis". In: *Proceedings of a symposium on Compiler optimization*. 1970, pp. 1–19. DOI: 10.1145/800028.808479.

[108] Lucas Davi, Patrick Koeberl, and Ahmad-reza Sadeghi. "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation". In: *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, June 2014, pp. 1–6. DOI: 10.1109/DAC.2014.6881460.

[109] Intel Corporation. *Control-flow Enforcement Technology Preview*. 2017. URL: http://intel.com/.%0Awww.intel.com/design/literature.htm..

[110] Martín Abadi et al. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security* 13.1 (Oct. 2009), pp. 1–40. DOI: 10.1145/1609956.1609960.

[111] *Control Flow Guard | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/windows/desktop/secbp/control-flow-guard.

[112] Arm. "Providing protection for complex software". In: (), pp. 1–25.

[113] Tao Zhang et al. "Anomalous path detection with hardware support". In: *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems - CASES '05*. New York, New York, USA: ACM Press, 2005, p. 43. DOI: 10.1145/1086297.1086305.

[114] Yubin Xia et al. "CFIMon: Detecting violation of control flow integrity using performance counters". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, June 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263958.

[115] Kui Xu et al. “Probabilistic Program Modeling for High-Precision Anomaly Classification”. In: *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, July 2015, pp. 497–511. DOI: 10.1109/CSF.2015.37.

[116] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. “Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2015, pp. 401–413. DOI: 10.1145/2810103.2813654.

[117] Ali Jose Mashtizadeh et al. “CCFI: Cryptographically Enforced Control Flow Integrity”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951. DOI: 10.1145/2810103.2813676.

[118] Hans Liljestrand et al. “PAC it Up: Towards pointer integrity using ARM pointer authentication”. In: *Proceedings of the 28th USENIX Security Symposium*. 2019, pp. 177–194.

[119] Jinfeng Li et al. “Zipper Stack: Shadow Stacks Without Shadow”. In: *European Symposium on Research in Computer Security*. Guildford, UK: Springer, 2020, pp. 338–358. DOI: 10.1007/978-3-030-58951-6{_}17.

[120] Qualcomm Product Security. “Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions”. In: (2017).

[121] Castro Miguel, Manuel Costa, and Tim Harris. “Securing Software by Enforcing Data-flow Integrity”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160.

[122] Thomas Nyman et al. “HardScope: Hardening Embedded Systems Against Data-Oriented Attacks”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. New York, NY, USA: ACM, June 2019, pp. 1–6. DOI: 10.1145/3316781.3317836.

[123] G. Edward Suh et al. “Secure program execution via dynamic information flow tracking”. In: *ACM Sigplan Notices*. Vol. 39. 11. 2004, pp. 85–96. DOI: 10.1145/1037949.1024404.

[124] James Newsome and Dawn Song. “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software”. In: *NDSS 05 Networks and Distributed Systems Security* (2005).

[125] Winnie Cheng et al. “TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting”. In: *11th IEEE Symposium on Computers and Communications (ISCC’06)*. IEEE, 2006, pp. 749–754. DOI: 10.1109/ISCC.2006.158.

[126] James Clause, Wanchun Li, and Alessandro Orso. “Dytan: A generic dynamic taint analysis framework”. In: *Proceedings of the 2007 international symposium on Software testing and analysis - ISSTA ’07*. New York, New York, USA: ACM Press, 2007, p. 196. DOI: 10.1145/1273463.1273490.

[127] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Technical Report)”. In: *Secure Systems Lab, Vienna ...* (2006).

[128] Steven Arzt et al. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". In: *Pldi* (2014), pp. 1–10. DOI: 10.1145/2594291.2594299.

[129] Marcelo Arroyo, Francisco Chiotta, and Francisco Bavera. "An user configurable clang static analyzer taint checker". In: *2016 35th International Conference of the Chilean Computer Science Society (SCCC)*. IEEE, Oct. 2016, pp. 1–12. DOI: 10.1109/SCCC.2016.7835996.

[130] Aravind Machiry et al. "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers". In: *26th USENIX Security Symposium*. 2017, pp. 1007–1024.

[131] Lars Ole Andersen. "Program analysis and specialization for the C programming language". PhD thesis. Citeseer, 1994.

[132] Crispin Cowan et al. "PointGuard™: Protecting pointers from buffer overflow vulnerabilities". In: *Proceedings of the 12th USENIX Security Symposium*. Washington, D.C.: USENIX Association, 2003, pp. 91–104.

[133] Sandeep Bhatkar and R. Sekar. "Data space randomization". In: *5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Paris, France: Springer, 2008, pp. 1–22.

[134] M.G. Matthew G. Schultz et al. "Data mining methods for detection of new malicious executables". In: (2001), pp. 38–49. DOI: 10.1109/SECPRI.2001.924286.

[135] T. Abou-Assaleh et al. "N-gram-based detection of new malicious code". In: *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* 2.1 (2004). DOI: 10.1109/CMPSAC.2004.1342667.

[136] T. Abou-Assaleh et al. "Detection of new malicious code using n-grams signatures". In: *Proceedings of Second Annual Conference on Privacy, Security and Trust* (2004), pp. 193–196. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.6349%5C&rep=rep1%5C&type=pdf.

[137] J Zico Kolter and Marcus a Maloof. "Learning to Detect and Classify Malicious Executables in the Wild". In: *Journal of Machine Learning Research* 7 (2006), pp. 2721–2744. DOI: 10.1145/1014052.1014105.

[138] T. Li, C. Zhang, and M. Ogihara. "A comparative study of feature selection and multiclass classification methods for tissue classification based on gene expression". In: *Bioinformatics* 20.15 (Oct. 2004), pp. 2429–2437. DOI: 10.1093/bioinformatics/bth267.

[139] D. Krishna Sandeep Reddy and Arun K. Pujari. "N-gram analysis for computer virus detection". In: *Journal in Computer Virology* 2.3 (Nov. 2006), pp. 231–239. DOI: 10.1007/s11416-006-0027-8.

[140] D Krishna Sandeep Reddy, Subrat Kumar Dash, and Arun K Pujari. "New malicious code detection using variable length n-grams". In: *Information Systems Security*. Springer, 2006, pp. 276–288.

[141] Igor Santos et al. “N-grams-based File Signatures for Malware Detection.” In: *Iceis (2)* (2009), pp. 317–320. URL: https://hexena.googlecode.com/files/penya%5C_ICEIS09%5C_N-grams-based%5C_File%5C_Signatures%5C_for%5C_Malware%5C_Detection.pdf.

[142] Zahra Salehi, Mahboobeh Ghiasi, and Ashkan Sami. “A miner for malware detection based on API function calls and their arguments”. In: *The 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012)*. IEEE, May 2012, pp. 563–568. DOI: 10.1109/AISP.2012.6313810.

[143] Dolly Uppal et al. “Malware detection and classification based on extraction of API sequences”. In: *2014 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. IEEE, Sept. 2014, pp. 2337–2342. DOI: 10.1109/ICACCI.2014.6968547.

[144] Vinod P. Nair et al. “MEDUSA: MEtamorphic malware Dynamic analysis using signature from API”. In: *Proceedings of the 3rd international conference on Security of information and networks - SIN ’10*. January. New York, New York, USA: ACM Press, 2010, p. 263. DOI: 10.1145/1854099.1854152.

[145] Radu S. Pirscoveanu et al. “Analysis of malware behavior: Type classification using machine learning”. In: *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*. IEEE, June 2015, pp. 1–7. DOI: 10.1109/CyberSA.2015.7166128.

[146] Steven Strandlund Hansen et al. “An approach for detection and family classification of malware based on behavioral analysis”. In: *2016 International Conference on Computing, Networking and Communications (ICNC)*. IEEE, Feb. 2016, pp. 1–5. DOI: 10.1109/ICCNC.2016.7440587.

[147] *Cuckoo Sandbox: Automated Malware Analysis*. URL: https://cuckoosandbox.org/.

[148] Ksenia Tsyganok et al. “Classification of polymorphic and metamorphic malware samples based on their behavior”. In: *Proceedings of the Fifth International Conference on Security of Information and Networks - SIN ’12*. New York, New York, USA: ACM Press, 2012, pp. 111–116. DOI: 10.1145/2388576.2388591.

[149] Davide Canali et al. “A quantitative study of accuracy in system call-based malware detection”. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*. New York, New York, USA: ACM Press, 2012, p. 122. DOI: 10.1145/2338965.2336768.

[150] Shaza Zeitouni et al. “ATRIUM: Runtime attestation resilient under memory attacks”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Nov. 2017, pp. 384–391. DOI: 10.1109/ICCAD.2017.8203803.

[151] Shuo Chen et al. “Non-control-data attacks are realistic threats”. In: *USENIX Security Symposium*. Vol. 5. USENIX Association, 2005, p. 146.

[152] Christoph Spang et al. “DExIE - An IoT-Class Hardware Monitor for Real-Time Fine-Grained Control-Flow Integrity”. In: *Journal of Signal Processing Systems* 94.7 (July 2022), pp. 739–752. DOI: 10.1007/s11265-021-01732-5.

[153] Atmel. *Customizable Microcontroller AT91CAP7S450A, AT91CAP7S250A*. URL: https://pdf1.alldatasheet.com/datasheet-pdf/view/255445/ATMEL/AT91CAP7S450A.html.

[154] Atmel. *Customizable Microcontroller AT91CAP7E*. URL: https://pdf1.alldatasheet.com/datasheet-pdf/view/257048/ATMEL/AT91CAP7E.html.

[155] *Atmel Announces CAP Customizable Microcontrollers | Berkeley Design Technology, Inc.* URL: https://www.bdti.com/InsideDSP/2007/07/18/atmel-announces-cap-customizable-microcontrollers.

[156] Richard W M Jones and Paul H J Kelly. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs". In: *Aadebug* (1997), pp. 13–26. DOI: 10.1.1.221.7393.

[157] Gary A. Kildall. "A unified approach to global program optimization". In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '73*. New York, New York, USA: ACM Press, Oct. 1973, pp. 194–206. DOI: 10.1145/512927.512945.

[158] M.R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*. IEEE, 2001, pp. 3–14. DOI: 10.1109/WWC.2001.990739.

[159] Hong Hu et al. "Automatic Generation of Data-Oriented Exploits". In: *USENIX Security Symposium*. 2015, pp. 177–192. DOI: 10.1109/SP.2014.25.

[160] Yaoqi Jia and Shuo Chen. "The " Web / Local " Boundary Is Fuzzy : A Security Study of Chrome ' s Process-based Sandboxing". In: (2016), pp. 791–804.

[161] *SSH CRC-32 Compensation Attack Detector Vulnerability*. URL: http://www.securityfocus.com/bid/2347.

[162] *Google Chrome CVE-2014-1705 Remote Code Execution Vulnerability*. URL: https://www.securityfocus.com/bid/66239.

[163] *Wu-Ftpd Remote Format String Stack Overwrite Vulnerability*. URL: https://www.securityfocus.com/bid/1387.

[164] *Todd Miller Sudo 'Sudo_Debug()' Path Resolution Local Privilege Escalation Vulnerability*. URL: https://www.securityfocus.com/bid/51719.

[165] *Null HTTPd Remote Heap Overflow Vulnerability*. URL: https://www.securityfocus.com/bid/5774.

[166] *FreeBSD 'telnetd' Daemon Remote Buffer Overflow Vulnerability*. URL: https://www.securityfocus.com/bid/51182/info.

[167] *ghttpd Daemon Buffer Overflow Vulnerability*. URL: https://www.securityfocus.com/bid/2879.

[168] Crispan Cowan et al. "StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks StackGuard : Automatic Adaptive Detection and Prevention of". In: *USENIX Security Symposium*. Vol. 98. USENIX Association, 1998, pp. 63–78.

[169] Lucas Davi et al. “HAFIX: Hardware-Assisted Flow Integrity Extension”. In: *Proceedings of the 52nd Annual Design Automation Conference*. New York, NY, USA: ACM, June 2015, pp. 1–6. DOI: 10.1145/2744769.2744847.

[170] Nathan Burow, Xinping Zhang, and Mathias Payer. “SoK: Shining light on shadow stacks”. In: *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, 2019, pp. 985–999. DOI: 10.1109/SP.2019.00076.

[171] Ben Hardekopf and Calvin Lin. “Flow-sensitive pointer analysis for millions of lines of code”. In: *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, Apr. 2011, pp. 289–298. DOI: 10.1109/CGO.2011.5764696.

[172] Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. “Unification-based Pointer Analysis without Oversharing”. In: *Proceedings of the 19th Conference on Formal Methods in Computer-Aided Design, FMCAD 2019* (June 2019), pp. 37–45. URL: http://arxiv.org/abs/1906.01706.

[173] Jean-Philippe Aumasson and Daniel J. Bernstein. “SipHash: A Fast Short-Input PRF”. In: *13th International Conference on Cryptology in India (INDOCRYPT 2012)*. Vol. 7668 LNCS. Kolkata, India: Springer Berlin Heidelberg, 2012, pp. 489–508. DOI: 10.1007/978-3-642-34931-7{_}28.

[174] David Floyer. *Exiting x86: Why Apple and Microsoft are embracing the Arm-based PC*. 2020. URL: https://siliconangle.com/2020/06/26/exiting-x86-apple-microsoft-embracing-arm-based-pc/.

[175] Grigori Fursin. *Collective Benchmark (cBench): collection of open-source programs and multiple datasets from the community*. URL: https://sourceforge.net/projects/cbenchmark/files/cBench/V1.1/.

[176] Guillaume Bonfante et al. “Control Flow Graphs as Malware Signatures”. In: *International workshop on the Theory of Computer Viruses*. 2007.

[177] Philip O’Kane, Sakir Sezer, and Kieran McLaughlin. “Obfuscation: The Hidden Malware”. In: *IEEE Security & Privacy* 9.5 (Sept. 2011), pp. 41–47. DOI: 10.1109/MSP.2011.98.

[178] Mark Hall et al. “The WEKA data mining software”. In: *ACM SIGKDD Explorations Newsletter* 11.1 (Nov. 2009), pp. 10–18. DOI: 10.1145/1656274.1656278.

[179] *11 of the worst ransomware - we name the internet nastiest extortion malware | Gallery | Computerworld UK*. URL: https://goo.gl/wNDoL4.

[180] *VirusTotal - Free Online Virus, Malware and URL Scanner*. URL: https://virustotal.com/.

[181] Mederic Hurier et al. “Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware”. In: *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, May 2017, pp. 425–435. DOI: 10.1109/MSR.2017.57.

[182] Marcos Sebestian et al. “AVclass: A Tool for Massive Malware Labeling”. In: *International Symposium on Research in Attacks, Intrusions and Defenses* (2016), pp. 230–253. DOI: 10.1007/978-3-319-11379-1.

[183] Sudhir Varma and Richard Simon. "Bias in error estimation when using cross-validation for model selection". In: *BMC Bioinformatics* 7.1 (Dec. 2006), p. 91. DOI: 10.1186/1471-2105-7-91.

[184] *TrendLabs Security Intelligence BlogPOWELIKS: Malware Hides In Windows Registry - TrendLabs Security Intelligence Blog*. URL: http://blog.trendmicro.com/trendlabs-security-intelligence/poweliks-malware-hides-in-windows-registry/.

[185] *Hunting the Mutex - Palo Alto Networks Blog*. URL: https://researchcenter.paloaltonetworks.com/2014/08/hunting-mutex/.

[186] Suleiman Y Yerima, Sakir Sezer, and Gavin McWilliams. "Analysis of Bayesian classification-based approaches for Android malware detection". In: *IET Information Security* 8.1 (Jan. 2014), pp. 25–36. DOI: 10.1049/iet-ifs.2013.0095.

[187] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. "Automated static code analysis for classifying android applications using machine learning". In: *Proceedings - 2010 International Conference on Computational Intelligence and Security, CIS 2010* (2010), pp. 329–333. DOI: 10.1109/CIS.2010.77.

[188] Ashkan Sami et al. "Malware detection based on mining API calls". In: *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*. New York, New York, USA: ACM Press, 2010, p. 1020. DOI: 10.1145/1774088.1774303.

[189] Munir Geden. "Ngram and Signature Based Malware Detection in Android Platforms". Master's thesis. University College London, 2015. URL: https://www.researchgate.net/publication/334262800_Ngram_and_Signature_Based_Malware_Detection_in_Android_Platform.

[190] Pengtao Zhang and Ying Tan. "Class-wise information gain". In: *2013 IEEE Third International Conference on Information Science and Technology (ICIST)*. IEEE, Mar. 2013, pp. 972–978. DOI: 10.1109/ICIST.2013.6747700.

[191] Nicholas D. Matsakis and Felix S. Klock. "The rust language". In: *ACM SIGAda Ada Letters* 34.3 (Nov. 2014), pp. 103–104. DOI: 10.1145/2692956.2663188.

Appendices

A
Appendix

A.1 Code Snippets of Real-world Vulnerabilities

```
void do_authentication(char *user, ...) {
    int authenticated = 0; ...
    while (!authenticated) {
        /* Get a packet from the client*/
        type = packet_read();
        switch (type) {
            ...
            case SSH_CMSG_AUTH_PASSWORD:
            if (auth_password(user, password))
                authenticated=1;
            case
            ...
        }
        if (authenticated) break;
    }
    /* Perform session preparation. */
    do_authenticated(pw);
}
```

(a) OpenSSH - CVE-2001-0144.

```
bool SecurityOrigin::canAccess
(const SecurityOrigin* other) const {
    if (m_universalAccess)
        return true;
    if (this == other)
        return true;
    if (isUnique() || other->isUnique())
        return false;
}
```

(b) Chrome - CVE-2014-1705.

```
struct passwd { uid_t pw_uid; ... } *pw;
...
int uid = getuid();
pw->pw_uid = uid;
... //format string error
void passive(void) { ...
    seteuid(0); //set root uid
    ...
    seteuid(pw->pw_uid); //set normal uid
    ...
}
```

(c) wu-ftpd - CVE-2000-0573.

```
struct user_details { uid_t uid; ... } ud;
... //run with root uid
ud.uid = getuid(); //in get_user_info()
...
vfprintf(...); //in sudo_debug()
...
setuid(ud.uid); //in sudo_askpass()
...
```

(d) sudo - CVE-2012-0809

```
void start_login(char *host,..) {
    addarg(&argv,loginprg);
    addarg(&argv,"-h");
    addarg(&argv,host);
    addarg(&argv,"-p");
    execv(loginprg,argv);
}
```

(e) Netkit Telnetd - CVE-2002-1496.

```
int serveconnection(int sockfd) {
    char *ptr; // pointer to the URL.
    // ESI is allocated // to this variable.
    ...
    if (strstr(ptr,"/.."))
        reject the request;
    log(...);
    if (strstr(ptr,"cgi-bin"))
        Handle CGI request ...
}
```

(f) Ghttpd - CVE-2001-820

Figure A.1: Code snippets forming basis to real world data-oriented attack scenarios.

A.2 Call Graphs of Bare-metal Examples

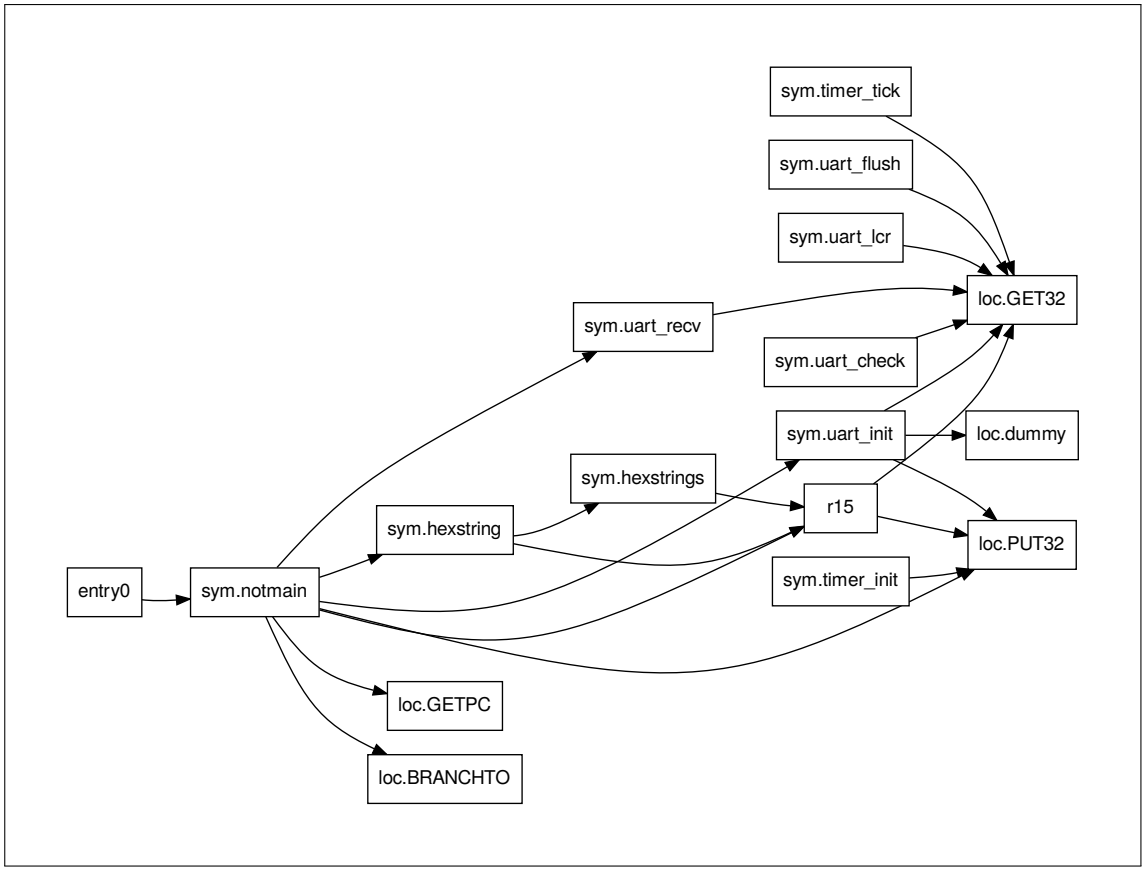


Figure A.2: Call graph of *bootloader* image.

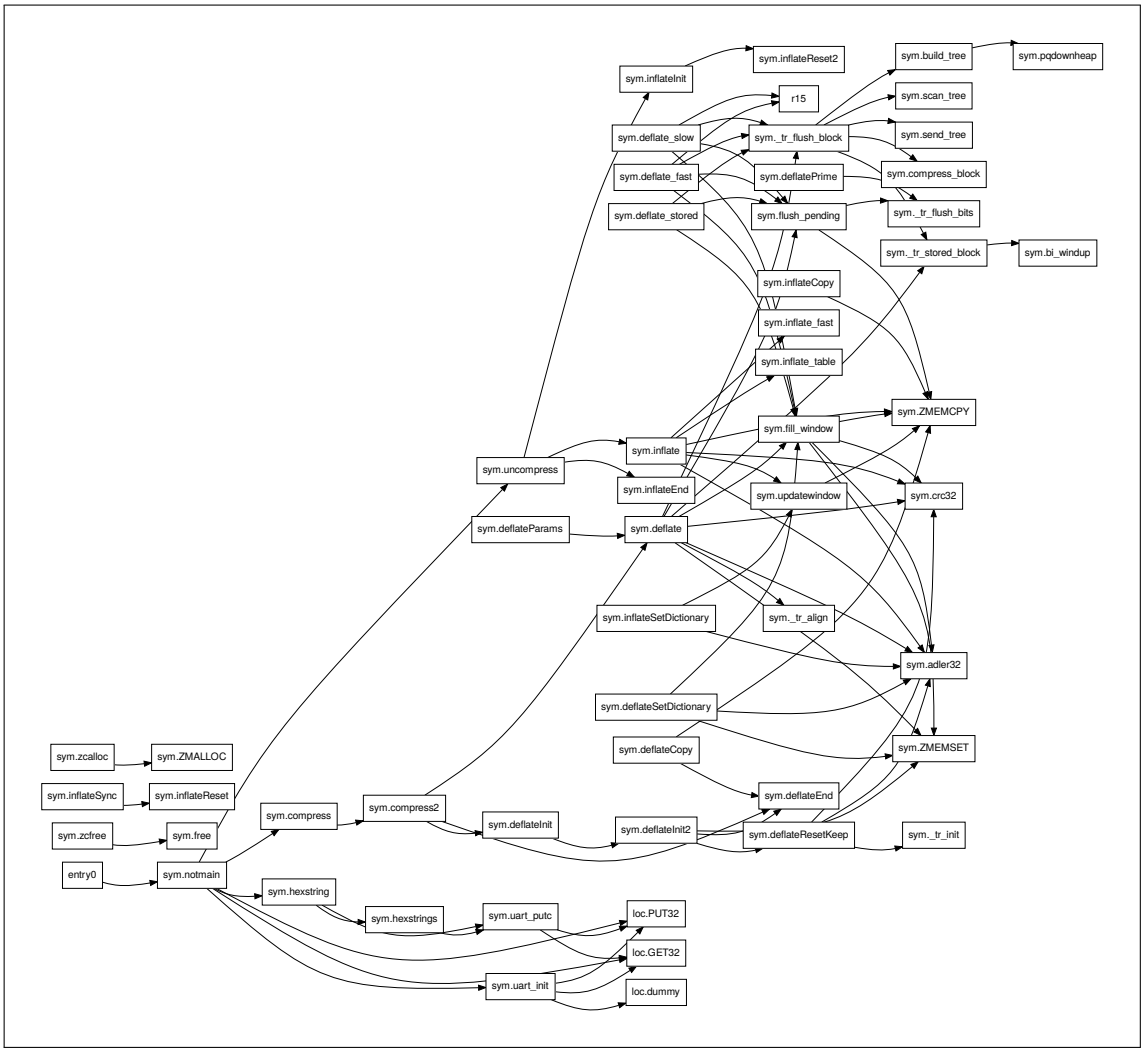


Figure A.3: Call graph of *zlib* library as the most complex bare-metal software examined.

A.3 JSON Structures of Cuckoo Reports

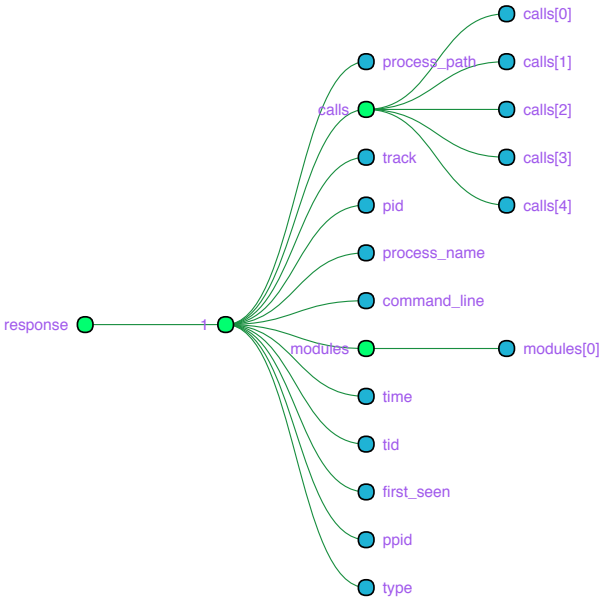


Figure A.4: JSON structure of an example process log.

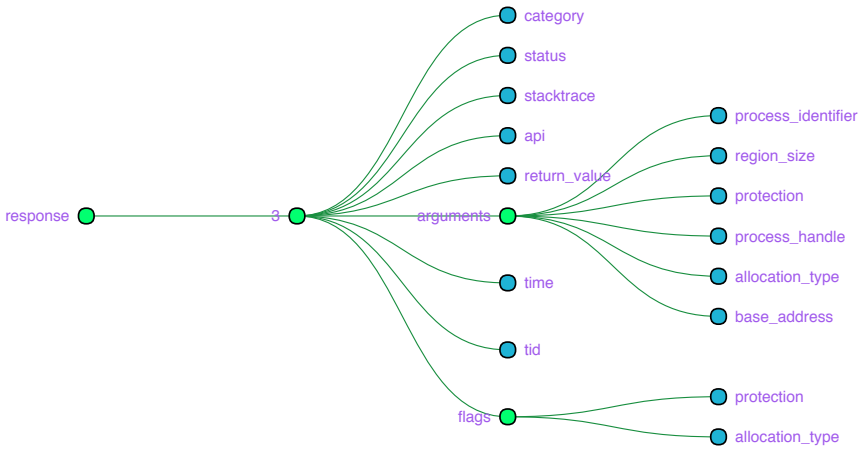


Figure A.5: JSON structure showing an example call trace.

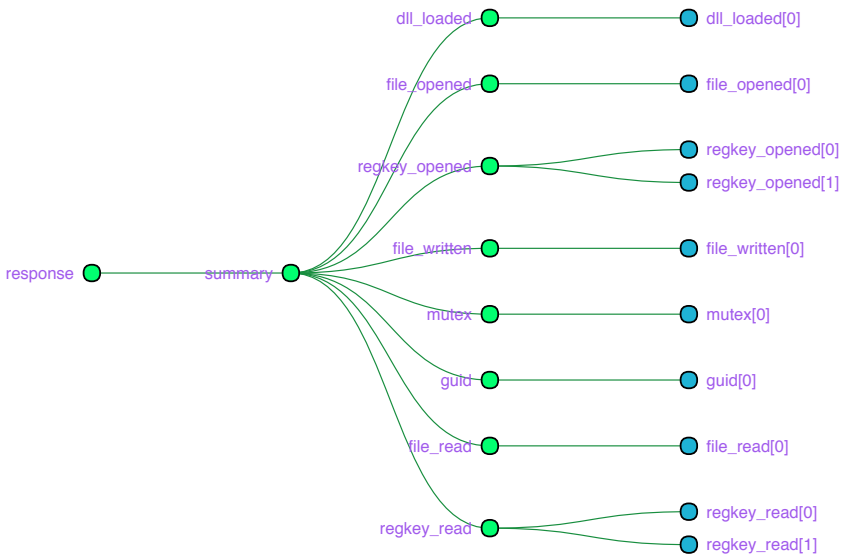


Figure A.6: JSON structure containing the behavioural artefacts such as files accesses, registry keys, DLLs loaded, mutexes.

Generated Cuckoo reports (1.4GB) of the collected samples can be found at: `https://goo.gl/e8jbXq`.
Source code of the framework is also available at `https://github.com/msgeden/familyclassifier`