FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs

DOI:
10.1145/3402937

Document Version
Accepted author manuscript

Citation for published version (APA):
La, T., Mätas, K., Grunchevski, N., Pham, K., & Koch, D. (2020). FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, *13*(3), [3402937]. https://doi.org/10.1145/3402937

Published in:
ACM Transactions on Reconfigurable Technology and Systems

FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs

TUAN MINH LA, KASPAR MATAS, NIKOLA GRUNCHEVSKI, KHOA DANG PHAM, and DIRK KOCH, The University of Manchester, UK

Sharing configuration bitstreams rather than netlists is a very desirable feature to protect IP or to share IP without longer CAD tool processing times. Furthermore, an increasing number of systems could hugely benefit from serving multiple users on the same FPGA, for example, for resource pooling in cloud infrastructures.

This paper researches the threat that a malicious application can impose on an FPGA based system in a multi-tenancy scenario from a hardware security point of view. In particular, this paper evaluates the risk systematically for FPGA power-hammering through short-circuits and self-oscillating circuits which potentially may cause harm to a system. This risk includes implementing, tuning, and evaluating all FPGA self-oscillators known from the literature, but also, developing a large number of new power-hammering designs which have not been considered before. Our experiments demonstrate that malicious circuits can be tuned to the point that just 3% of the logic available on an Ultra96 FPGA board can draw the power budget of the entire FPGA board. This fact suggests a waste power potential for datacenter FPGAs in the range of kilowatts.

In addition to carefully analyzing FPGA hardware security threats, we present the FPGA virus scanner FPGADefender that can detect (possibly) any self-oscillating FPGA circuit, as well as detecting short-circuits, high fanout nets, and a tapping onto signals outside the scope of a module for protecting data center FPGAs such as Xilinx UltraScale+ devices at the bitstream level.

CCS Concepts: • **Security and privacy → Side-channel analysis and countermeasures**; **Hardware attacks and countermeasures**; • **Hardware → Reconfigurable logic and FPGAs**.

Additional Key Words and Phrases: cloud computing, hardware security, FPGA, denial-of-service, power-hammering, side-channel, bitstream, mitigation, countermeasure

ACM Reference Format:
Tuan Minh La, Kaspar Matas, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. 2018. FPGADefender: Malicious Self-Oscillator Scanning for Xilinx UltraScale+ FPGAs. *ACM Trans. Graph.* 37, 4, Article 111 (August 2018), 31 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

With the present trend of FPGAs being offered in cloud data centers [24, 25, 53, 68] and also more and more hardware designs being provided as bitstreams (e.g., [10, 66]), there is a strong need to investigate FPGA hardware security. While traditionally, the FPGA hardware, as well as the design running on the FPGA, was entirely designed and integrated by one party, we are now increasingly heading towards ecosystems with more complex supply chains. For example, nowadays we have cloud data centers using off-the-shelf FPGA boards as well as a stack of own, vendor, and third

Authors' address: Tuan Minh La, tuan.la@postgrad.manchester.ac.uk; Kaspar Matas, kaspar.matas@manchester.ac.uk; Nikola Grunchevski, nikola.grunchevski@manchester.ac.uk; Khoa Dang Pham, khoa.pham@manchester.ac.uk; Dirk Koch, dirk.koch@manchester.ac.uk, The University of Manchester, Manchester, UK.

0730-0301/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

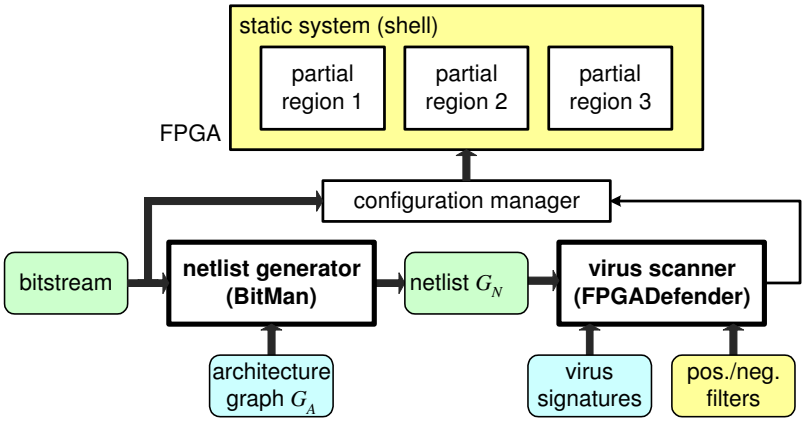


Fig. 1. Envisioned system with a virus scanner for detecting malicious configuration bitstreams.

party IPs and tools in order to provide users with FPGA-enabled Acceleration-as-a-Service (AaaS) offerings [24] as well as with FPGA-as-a-Service (FaaS) offerings [25, 68].

The origins of this trend can be referred to as the *FPGA cloud effect*, which has stimulated a large body of FPGA-related research, which was triggered by the announcement of major cloud service providers to offer FPGA instances (e.g., Amazon AWS with its F1 FPGA instances). For the first time, these offerings gave everybody the possibility to access high-end FPGA hardware at low initial cost and without the need to install any FPGA design tools locally. This effect also initiated a Renaissance in FPGA hardware security research [47], and several recent papers have demonstrated attacks on 1) how to compromise the integrity of a system and how to deny a system's service and 2) how to leak information from other parts of the system, as surveyed in Section 2.

However, while there are many attacks demonstrated, there are only a few papers published that propose countermeasures for FPGA hardware security threats. Cloud service providers such as Amazon, Alibaba, Baidu, and Nimbix rely entirely on the FPGA vendor Xilinx to protect their FPGA infrastructure by using design rule checking (DRC) at the netlist level [24, 25, 27, 28]. We would like to stress that the common approach of AWS and other FPGA cloud service providers is insufficient for adding security to their system because the (Xilinx) DRCs are only catching LUT-based ROs (see [19, 57], and our examples in Section 3). This lack of countermeasures is surprising as security will be a paramount requirement for the possible next wave of systems where multiple tenants may share the same FPGA in an FaaS setting (which allows harnessing the full cloud advantages for processing and resource pooling also for FPGAs) or where end users may download and execute configuration bitstreams as kind of hardware apps. Please note that existing FPGA cloud offerings can already be considered as a multi-tenant scenario consisting of a shell with the I/O infrastructure (commonly provided by the cloud service provider) and the user modules that interface to a server node using that shell. In this setting, a user should not be able to interfere with the shell in an uncontrolled manner, and it is essential to prevent a scenario that would allow a hijacking of any shell functionality (e.g., for gaining access to the PCIe core which is connected to the server host node [26]).

In this paper, we propose an FPGA virus scanner, named FPGADefender, that scans partial module bitstreams such that a system can reject malicious modules if needed. We limit ourselves to partial bitstreams because the present version of FPGADefender does only fully support logic tiles (CLBs), on-chip memories (BRAMs) and arithmetic blocks (DSPs). However, that is sufficient for data center applications where users access all peripherals (e.g., DDR memory and PCIe) through a

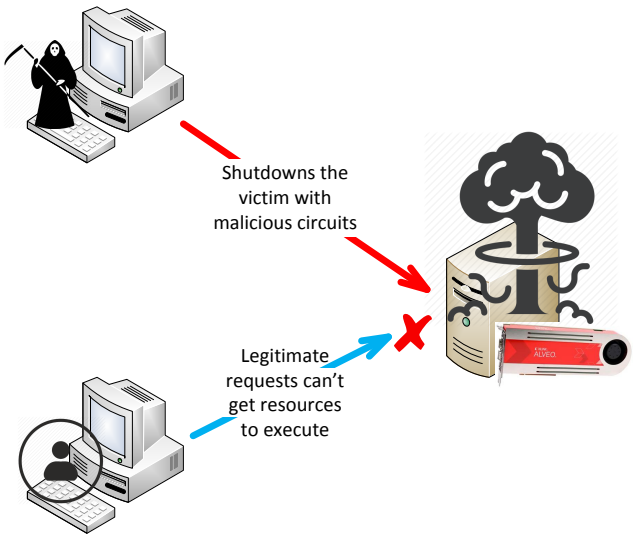


Fig. 2. An illustration of the denial-of-service-like (DoS-like) threat model. A user may try to shutdown an FPGA service in a data center by sending malicious circuits such that legitimate requests from other users cannot use the FPGA resources. Short-circuits and power-hammering designs can be utilized for such attacks on the system availability. Furthermore, this kind of attack may potentially age or damage the equipment.

shell and where direct access to I/O pins is commonly prohibited. The envisioned system featuring FPGA virus scanner is shown in Figure 1. In a traditional system, a (partial) bitstream would be sent directly to a configuration manager that is in charge of pumping the bitstream binary into the fabric. To perform virus scanning from a bitstream, we first have to rebuild a netlist, which in turn requires an FPGA architecture model. This netlist can then be scanned by the actual virus scanner engine, which requires virus definitions (also known as virus signatures) and system-specific constraints. If the scanner detects a malicious construct in the bitstream, this will be flagged to the configuration manager, which may reject malicious bitstreams.

The current most severe FPGA hardware security threats relate to self-oscillators. Self-oscillators (SOs) are circuits where the oscillation does not depend on an external clock (e.g., from a quartz crystal) but on some feedback implemented in the soft logic of an FPGA. Ring-oscillators (ROs) are a subset of self-oscillators representing circuits that are based on inverting combinatorial feedback loops. To provide the best possible protection against self-oscillators, we examined a large number of reported as well as several novel self-oscillating designs. We quantified their potential threats to power-hammering thoroughly and included virus signatures for all of them to FPGADefender.

Throughout the rest of the paper, we will describe the concepts, implementation, and evaluation of the virus scanning in more detail. Our key contributions are:

- An in-depth study on FPGA self-oscillators, including the discovery of novel oscillator designs (Section 3).
- A model for FPGA virus scanning and virus signatures for detecting oscillators, wire-tapping, and short-circuits from a bitstream (Section 4).
- An implementation, evaluation, and discussion of FPGADefender (Section 5).

In addition to these critical contributions, we are providing an overview of FPGA security threats (Section 2) and a comparison of software virus scanning versus hardware virus scanning (Section 4.1). Note that this paper uses the term *virus scanning* in its figurative meaning for detecting all kinds of

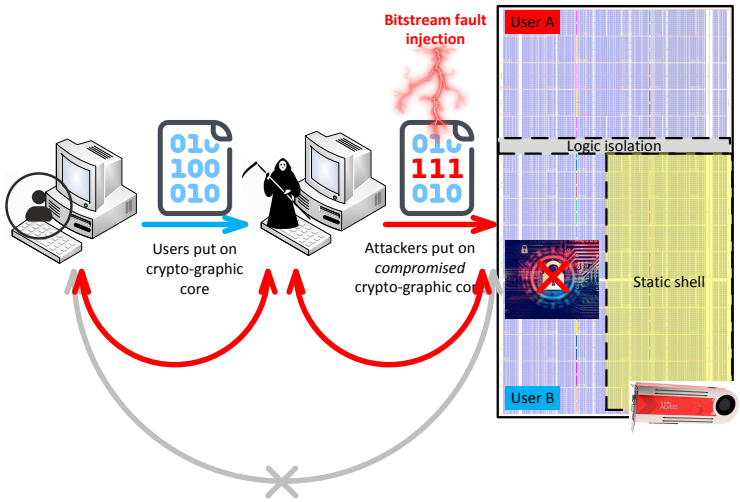


Fig. 3. An illustration of the *bitstream fault injection* threat model in a multi-tenant computing environment.

malicious threats rather than in its original meaning of infecting a program with malicious code to spread out in a virus-like manner.

2 HARDWARE SECURITY THREATS FOR MULTI-TENANT FPGAS

Traditionally, FPGA industry vendors considered that the security of an FPGA was primary about protecting designs in terms of intellectual property (IP) in configuration data (or bitstream) against cloning/overbuilding, reverse engineering, tampering, and spoofing, as summarized in [63]. On the other hand, FPGAs are now integrated into data centers and cloud computing infrastructures [17, 24, 53], and hence, multi-tenant scenarios are expected to provide better utilization and power efficiency as compared to the current one-user-per-fabric scheme [65]. However, due to their deep low-level programmability, FPGAs comprise new threat models far beyond what is commonly known from conventional CPU/GPU systems. For instance, modules running on an FPGA may include circuits being able to measure system states at high accuracy, which may open physical side-channels to leak sensitive data from other users [18, 55] that are not available in known software threat models.

In this section, we take a brief literature review on potential threats against multi-tenant FPGAs which can be categorized into 1) attacks on the system availability (DoS-like attacks), 2) attacks on the system integrity (via bitstream fault injection), and 3) attacks on the user confidentiality (via physical side-channel analysis); as well as state-of-the-art countermeasures.

2.1 Attacks on the System Availability

Denial-of-service-like (DoS-like) attacks are used to bring down operating infrastructures or to compromise states in other system components that stay outside the scope of an attacking module. At the electrical level, two means for DoS-like attacks had been utilized: short-circuits and power-hammering.

Short-circuits on FPGAs have a long history [2, 3, 22]. In contrast, more recent research [6] demonstrated short-circuits within the multiplexers inside a switch matrix using a manipulated configuration bitstream resulting in a substantial current increase (with several *mA* extra current for a single routing multiplexer). Short-circuits may potentially age or even damage an FPGA chip

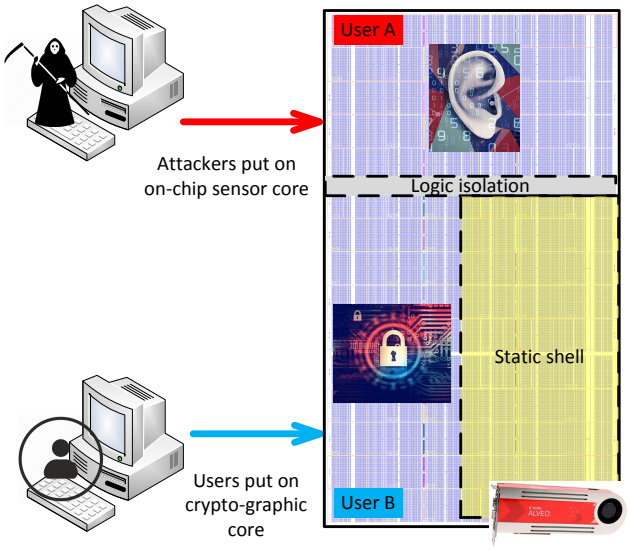


Fig. 4. An illustration of the *eavesdropping* threat model of user confidentiality in a multi-tenant computing environment.

permanently. Alternatively, this may be used for power-hammering, as discussed in the following paragraph.

Power-hammering is another mechanism to carry out DoS-like attacks. All current power-hammering attacks [20] are based on fast toggling circuits in order to draw a substantial amount of dynamic power. As we will show in Section 3, it is possible to implement self-oscillating circuits running in the GHz frequency domain with a corresponding dynamic power footprint. In [20], a grid of ring-oscillators could be activated at an adjustable rate (to stimulate resonance effects in the power supply regulation circuit). With this, several FPGA platforms such as Xilinx Virtex 6, Kintex 7, and Zynq-7000 FPGAs had been crashed (and in some cases requiring power-cycling for bringing up boards back into service). Although ring-oscillators are usually flagged with a warning by the vendor design tool flows and hence, are not allowed to be deployed on common cloud or data center infrastructures, recent research [19] has reported new ring-oscillator designs which can bypass such a Design Rule Checking (DRC).

In this paper, we will investigate a large number of known as well as newly developed oscillator variants that can be built from FPGA primitives, including LUTs, carry logics, and DSPs.

2.2 Attacks on the System Integrity

Further, FPGA security vulnerabilities include bitstream fault injection attacks [1, 58–60]. These types of attacks are man-in-the-middle attack variants that directly compromise the integrity of a system, commonly intending to leak a user's confidential data, as illustrated in Figure 3. However, bitstream fault injection attacks usually result in *invalid* FPGA bitstreams, which may not be easily deployed in data centers thanks to tamper-resistant features in modern FPGAs [13, 29]. Indeed, no such man-in-the-middle attack has been reported to be launched successfully against multi-tenant computing infrastructures. Attestation protocols would typically identify modifications to a bitstream. For example, a bitstream can be hashed before or after being loaded to the FPGA, and a hash mismatch would flag any modification. In the latter case, where configuration readback

is commonly used, FPGADefender would ensure that no bitstream would reach the FPGA that contains malicious configurations at the electrical level.

2.3 Attacks on the User Confidentiality

Side-channel attacks on FPGAs can be either active (e.g., timing fault injection) or passive (e.g., power analysis, crosstalk coupling, electromagnetic analysis, and thermal channel leakage). Differential Fault Analysis (DFA) was shown in [9] for breaking cryptographical implementations, whereas in [41], timing faults have been injected through a large number of ring-oscillators to cause voltage drops followed by analyzing the resulting faulty ciphertext using DFA for successfully revealing the secret key of a crypto-core.

To deploy a passive side-channel attack remotely, an attacker needs to measure parameters of covert channels such as latency, temperature, IR drop, or any crosstalk effect, as summarized in a recent survey [47]. Power analysis attacks have been demonstrated to leak the secret key of a cryptographic function that was running on the same FPGA [55], running on a CPU embedded on the same FPGA die [72], and running on a different FPGA on the same FPGA board [56]. All these attacks have in common that they use ring-oscillators to measure key-dependent fluctuations on the voltage.

In addition to sensing voltage, self-oscillators can be used to monitor latency variations induced from crosstalk effects [18, 19, 54]. In these studies, it was found that a wire carrying a logical one will slow down a ring-oscillator that is implemented directly adjacent to this wire. Therefore, by taking advantage of the sensitivity of self-oscillators, attackers can leak the current state of a signal, which is a concern in shared FPGA infrastructures. Furthermore, self-oscillators can be used as a sensor to collect system states stealthily [73].

In summary, it is of paramount importance to detect *any* self-oscillating circuit in the valid FPGA bitstreams, as enabled by FPGADefender.

2.4 State-of-the-art Countermeasures

To prevent side-channel power analysis attacks, different *masking* and *hiding* strategies had been proposed. In the masking strategy, an implementation of a cryptographic algorithm is transformed into another (typically larger) variant, which is functionally equivalent, but where the new circuit can remain secure, although the attacker can observe some details of the operation through a side-channel, as proposed in [35]. For example, a cryptographic module may change specific details of the implementation (e.g., if an S-box lookup in an AES implementation is performed sequentially or in parallel or if that S-box lookup is implemented through a table or some Boolean logic functions). This approach makes power analysis attacks much harder as the data leaked has additionally to be correlated with the implementation changing scheme used inside the secured core.

On the other hand, the hiding strategy aims at lowering the Signal Noise Ratio (SNR) during the operation by either adding more sources of noise or lessening the strength of the signal power trace that relates to the operation of the core, as suggested in [14, 21, 37, 69].

Ring-oscillators can be used to monitor the healthiness of an FPGA fabric [73] or even detect voltage drop attacks (e.g., power-hammering, power analysis) [52, 74]. Further, recent work has suggested using ring-oscillators not only to monitor a power analysis attack but also to respond against the attack by triggering some noise generators [40].

Although these works claimed to respond against power-hammering and power analysis attacks, it would be ideal that those kinds of attacks could be prevented and not even reach to an FPGA in the first place! This protection scheme also simplifies system management as it would otherwise require an exception handling after a fault due to an attack may have occurred. Therefore, checking

Table 1. Configurable resources of Ultra96 compared with the data center FPGA Alveo U200

Primitive count	ZU3EG	Alveo U200
CLB LUTs	70560	1182240
CLB flip-flops	141120	2364480
DSP Slices	360	6840
BRAM Slices	648	6480

an FPGA bitstream *before* deploying it on a multi-tenant FPGA is a desirable (if not essential) feature.

In a related work [42], LUT-based ring-oscillator designs are detected directly inside configuration bitstreams. While that work fundamentally showed that oscillator circuits could be detected from bitstreams, it was only shown for basic LUT-based oscillators (which are already spotted by the FPGA vendor tools). This limitation leaves an attacker the chance to deploy alternative oscillator designs (e.g., based on glitch amplification), which the work in [42] cannot identify. In contrast, FPGADefender is designed to detect any self-oscillating circuits in any user design. With *user design*, we refer to designs made of logic cells (LUTs), block RAMs, and DSP blocks only. This is the model used by all cloud service providers where users can only program these blocks while a static shell provides all other primitives that are commonly needed (e.g., for I/O and clocking). Furthermore, [42] was implemented on a Lattice FPGA, and those FPGAs are relatively small for building a multi-tenancy system. However, the vast majority of systems that would benefit from an FPGA virus scanner are based on modern FPGA architectures, which are substantially more complex (e.g., fracturable LUTs, complex DSP blocks with ALU functionality, complex clock networks, and a hierarchical routing fabric). Therefore, FPGADefender was designed to be compatible with Xilinx UltraScale+ FPGAs. This feature makes FPGADefender applicable in real-world data centers and cloud computing infrastructures [17, 24, 25].

In conclusion, discussed attacks compromise not only the user confidentiality but also the system availability and the system integrity of an FPGA infrastructure. Although previous countermeasures suggested that self-oscillators can also be used to detect potential power attacks or monitor system healthiness after the malicious bitstream is already loaded, it is crucial to have a stronger mechanism to detect the use of malicious circuits *before* it has already been deployed, as provided by FPGADefender for Xilinx UltraScale+ FPGAs.

3 A STUDY ON FPGA SELF-OSCILLATORS

In this section, we will provide an in-depth study on a wide range of self-oscillating circuits to quantity their potential threats with a focus on power-hammering. Besides, we use this study to tune FPGADefender for detecting all kinds of self-oscillating circuits that are practically implementable on Xilinx UltraScale+ FPGAs using any logic, memory, or arithmetic primitives. This insight is essential as *FPGADefender should not only detect some oscillator designs but any known designs in a user circuit.* Different effects can be used to design self-oscillators, as discussed in the next paragraphs. Because the actual oscillator speed depends on the supply voltage and temperature (which, in turn, relates to the current operation state of the FPGA), any oscillator is probably a potential path for a side channel. Therefore, even focusing on power-hammering in this section, by preventing oscillators, we will further prevent the most critical side channels that are deployable remotely.

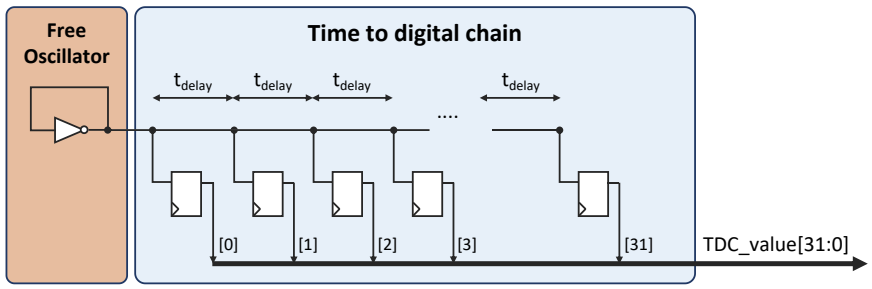


Fig. 5. Time To Digital construction.

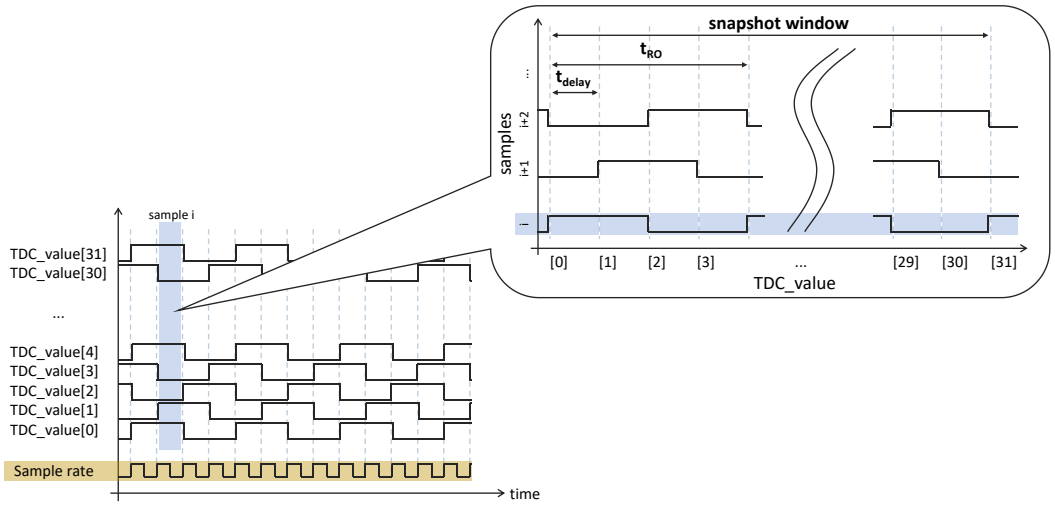


Fig. 6. Time-to-Digital Converter (TDC) waveform.

3.1 Experimental Setup

Our experiments are conducted on an Ultra96 board, which is equipped with a Xilinx Zynq UltraScale+ MPSoC ZU3EG. The primitive resources count for Ultra96 in comparison with a data center Alveo U200 [28] is shown in Table 1. We created 15 different Ring Oscillator (RO) variants and evaluated them in Table 2. To generate the RO circuits, the `PathSearch` function of the GoAhead tool [7] was used. That tool can perform a breadth-first search between arbitrary ports of the FPGA fabric and rank the resulting paths by their latency. The expected frequency is based on timing reports generated by the Xilinx Vivado tool [30], and the measured frequency on the FPGA is determined by using a Time-to-Digital Converter (TDC), as shown in Figure 5. Our TDC is a delay chain that allows us to take a snapshot of a signal propagating down the chain precisely. By using $N_{FF} = 32$ flip-flops (FFs) and $t_{delay} \approx 70ps$ (between two adjacent sample flip-flops), we can capture a signal with a *snapshot window* of $\approx 2170ps$ (Equation 1) and with a resolution of $70ps$ approximately (see Figure 6 for details). This latency corresponds to a frequency range from $246MHz$ to $7142MHz$ (see Equation 2 where N_{HIGH} and N_{LOW} are the number of consecutive FFs that have registered *HIGH*-state and *LOW*-state respectively). Figure 6 shows how the samples of a TDC are read out to measure a frequency.

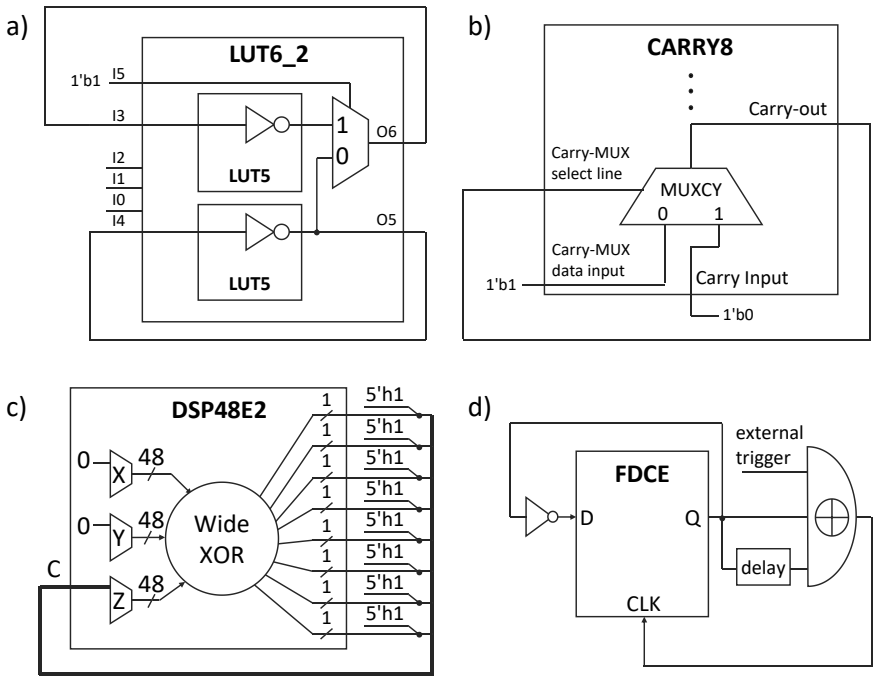


Fig. 7. a) Dual-RO from LUT6 primitive; b) RO design from Carry Logic; c) RO design from DSP; d) Glitch amplification.

It should be noted that the clock buffer primitives inside a programmable logic (PL) region of the FPGA fabric are rated for a maximum clock frequency of $891MHz$ [34]. Therefore, in order to ensure stable TDC measurements, we operate the TDC sampling FFs at a moderate clock frequency of $100MHz$. This is a difference to other clock measurement designs used for FPGA side-channel attacks, which feed the RO's output directly to clock inputs of some FFs to form a counter [18, 19, 54]. Because we aim for generating frequencies in the GHz regime, simple counter designs cannot be used.

$$snapshot\_window = N_{FF} \times t_{delay} \tag{1}$$

$$\begin{aligned} f_{RO} &= \frac{1}{t_{RO}} = \frac{1}{2 \times cycle\_path\_delay} \\ &\approx \frac{1}{t_{delay} \times (N_{HIGH} + N_{LOW})} \end{aligned} \tag{2}$$

TDCs are subject to temperature changes, and we used heater circuits (circuits that waste power) to heat the chip to 90°C before actually taking any measurement. The temperature is within the maximum operating temperature of the FPGA, which is 100°C [34]. The heaters are not running during the short period of time to take the (typically below one ms) measurements, and we use the temperature sensor that is built into the FPGA to implement the temperature control. Additionally, we took the median from 1000 measurements for each frequency reported in order to reduce the impact of quantization errors and noise.

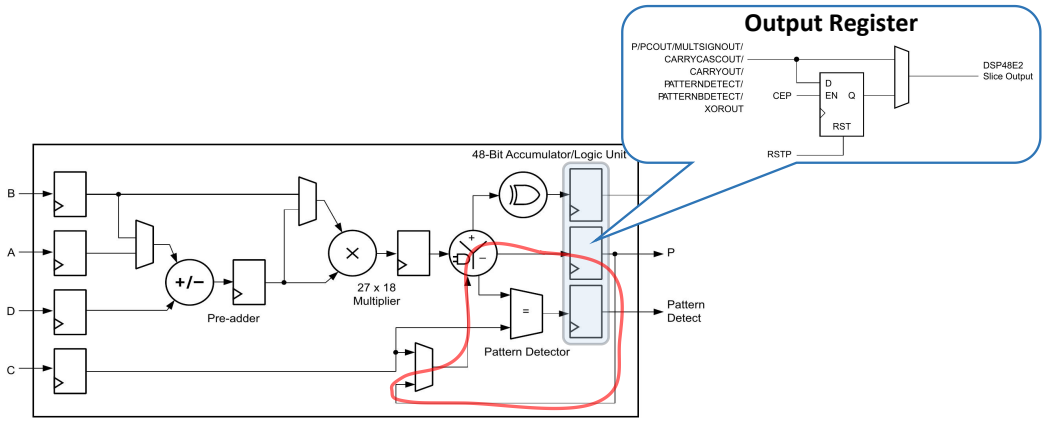


Fig. 8. Tentative internal combinatorial loop inside DSP. This figure is adopted from [32].

3.2 Combinatorial Self-Oscillator Variants

The simplest self-oscillator is a combinatorial loop, which is a circuit that consists of an odd number of chained inverters. We will refer to this basic oscillator as Ring-Oscillator (RO). The frequency of an RO can be calculated by the propagation delay of the whole combinatorial loop which includes propagation delay of all logic elements t_{logic} (e.g., LUTs or DSP blocks) and the net delay t_{net} (i.e. the routing delay) as given in Equation 3.

$$f_{RO} = \frac{1}{t_{RO}} = \frac{1}{2 \times (t_{logic} + t_{net})} \tag{3}$$

We performed a literature review on RO designs and found that previous researches [18–20, 41, 57] only use LUT primitives to implement ROs. To capture *any possible oscillator design*, we analyzed the exact internal architectures of the logic (i.e. SliceL/SliceM), arithmetic (i.e. DSP48E2), and BRAM primitives available in UltraScale+ FPGAs. And for each of these primitives, we asked the question *if there exists a configurable combinatorial path from any of the primary inputs to any of the primary outputs* because this is a fundamental requirement for designing ROs. This study has to incorporate all the different modes each primitive can be configured, and we found:

- **Slices**: We examined known RO designs through LUTs [18–20, 41, 57] as well as transparent latches [19, 42, 57] and self-oscillating circuits based on glitch amplification or asynchronous reset/preset [19, 42, 57]. In addition to these designs using FPGA slices, we found combinatorial paths that have not been previously reported by the community, but that can be used for ROs including 1) paths through *MUX primitives* (i.e. *F7Mux* and *F8Mux* multiplexers) inside the slices and 2) paths through the *carry logic* (i.e. the *Carry Look Ahead (CLA)* logic introduced in UltraScale+ FPGAs) (see Figure 7b).
- **DSPs** had not been considered in previous research for building oscillators. However, DSPs can be used in many different configuration options, and there are many possibilities for designing ROs. This is possible because DSP blocks can be used purely combinatorial without any pipeline registers between the primary inputs and outputs that would prevent self-oscillation. For instance, an RO can be formed by feeding the output of a DSP primitive back to an input for implementing a counter without using any register in the feedback path. The DSP48E2 primitives include not only multipliers but a tiny ALU that can perform bit-level operations that execute faster than arithmetic operations, and for the remainder of this paper,

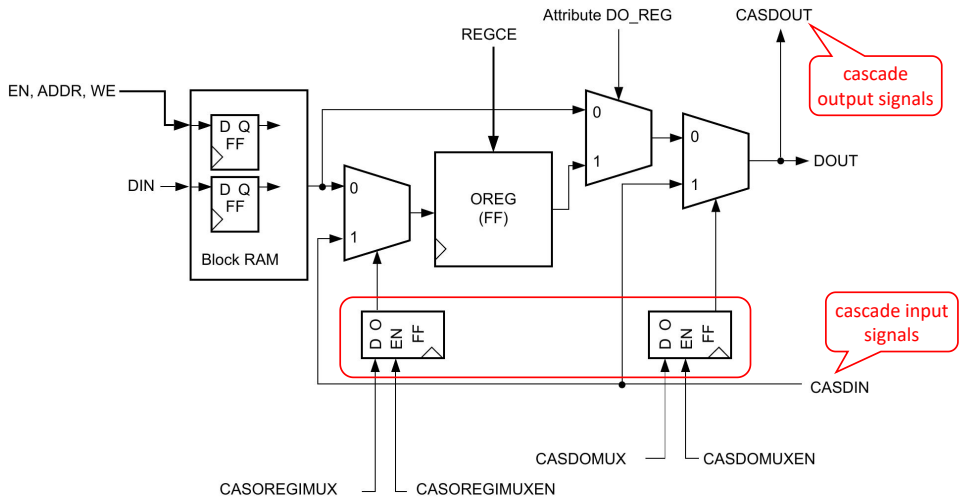


Fig. 9. BRAM cascade functional diagram. This figure is adopted from [33].

we will only report results for the wide-XOR instruction that was found oscillating the fastest (see also Figure 7c).

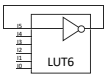
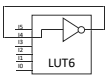
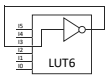
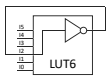
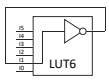
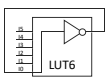
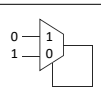
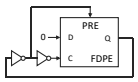
It is worth mentioning that we tried building an internal loop inside the DSPs, which may exist in accumulator mode. We investigated this path because the accumulator register can be bypassed to the output (as shown in Figure 8), and the documentation [32] does not state if the bypass may eventually be used together with the accumulator mode. However, we have not detected any switching activity or abnormal increase in power when configuring this option. This fact implies that the flip-flop output is fed back to the accumulator input rather than the output of the bypass multiplexer (see the top right box in Figure 8). Thus, the DSP accumulator mode can be considered as secure from possible cycles in DSP48E2 primitives.

- **BRAMs** are mostly comprised of synchronous components (i.e. memory cells) that are working on a clock basis with synchronous reset signals [33]. With this, we cannot implement any ROs directly through internal BRAM components. The only existing combinatorial part that we found is located inside the cascading logic, which is used to build larger memories from multiple consecutive BRAM primitives. However, the cascading chains have dedicated bottom-up routing resources that cannot be controlled by user logic, and cascade multiplexers are controlled by flip-flops, as shown in Figure 9. Therefore, BRAMs are considered to be RO-free.

We like to stress that most new oscillator designs do not throw any DRC critical warning/error message in the vendor tool Xilinx Vivado 2019.1, which means that these oscillators are possibly deployable, for example, on Amazon F1 cloud instances [57]. Table 2 provides an overview of most oscillator designs examined in this paper.

While there are papers discussing self-oscillators for FPGAs [19, 42, 57], we have not found a comprehensive study on performance tuning for improving the power-hammering potential as well as a corresponding evaluation of such oscillators on real FPGA hardware. For differential power analysis (DPA) attacks, an attacker typically seeks the fastest oscillator. In contrast, for a denial-of-service attack, the waste power efficiency (power drawn per unit resources) is more important. Even for a basic RO using LUT primitives, we found that the different LUT6 primitives inside a CLB (i.e. a cluster of 8 LUT6 primitives sharing a switch matrix) as well as using different LUT

Table 2. Variants of self-oscillating circuits studied on Xilinx UltraScale+ FPGAs. The results of power consumption are measured on the Ultra96 platform equipping with a Zynq UltraScale+ MPSoC ZU3EG.

No	Variants	Schematics	Number of loops	Loop Type	DRC Warning	Report Net Delay	Report Primitive Delay	Expected Frequency	Measured Frequency	Power	WPP
0	Empty design	ø	ø	ø	ø	ø	ø	ø	ø	2.94W	ø
1	RO using LUT6 (I5)		2000	Comb	✓ (LUTLP-1**)	49ps	41ps	5556MHz	5882MHz	7.32W (+4.38W)	26.63
2	RO using LUT6 (I4)		2000	Comb	✓ (LUTLP-1**)	51ps	66ps	4274 MHz	3937 MHz	6.84W (+3.90W)	23.69
3	RO using LUT6 (I3)		2000	Comb	✓ (LUTLP-1**)	46ps	100ps	3425MHz	3012MHz	5.99W (+3.05W)	18.52
4	RO using LUT6 (I2)		2000	Comb	✓ (LUTLP-1**)	50ps	116ps	3012MHz	2488MHz	5.63W (+2.68W)	16.32
5	RO using LUT6 (I1)		2000	Comb	✓ (LUTLP-1**)	62ps	150ps	2358MHz	2320MHz	6.59W (+3.65W)	22.21
6	RO using LUT6 (I0)		2000	Comb	✓ (LUTLP-1**)	71ps	177ps	2016MHz	1927MHz	6.35W (+3.41W)	20.75
7	Dual-RO from LUT6 primitive	Refer to Figure 7a	2000×2	Comb	✓ (LUTLP-1**)	O5: 308ps O6: 54ps	O5: 85ps O6: 100ps	O5: 1272MHz O6: 3247MHz	O5: 1235MHz O6: 2439MHz	8.04W (+5.10W)	31.00
8	Enhanced ROs design for power-hammering using high fanout to waste power on routing resources	Refer to Figure 10	2000	Comb	✓ (LUTLP-1**)	64ps	66ps	3846MHz	1779MHz	9.61W (+6.66W)	40.54
9	RO using MUX7		2000	Comb	✗	353ps	112ps	1075MHz	1126MHz	5.01W (+2.07W)	6.30
10	RO using MUX8		2000	Comb	✗	211ps	109ps	1563MHz	1681MHz	4.04W (+1.10W)	1.67
11	RO using Carry Logic	Refer to Figure 7b	2000	Comb	✗	381ps	104ps	1031MHz	1109MHz	5.14W (+2.19W)	1.67
12	RO using DSP	Refer to Figure 7c	360x8	Comb	✓ (DPIP-2*, DPOP-3*)	251ps	994ps	402MHz	585MHz	4.53W (+1.59W)	0.27
13	RO using latch [42, 57]		2000	Non-Comb	✗	173ps	96ps	1859MHz	1706MHz	5.14W (+2.19W)	13.35
14	RO using flip-flop [19, 42]		2000	Non-Comb	✓ (PDRC-153*, PLHOLDVIO-2*)	✗	✗	✗	555MHz	5.26W (+2.32W)	7.05
15	Glitch amplification	Refer to Figure 7d	2000	Non-Comb	✓ (PDRC-153*, PLHOLDVIO-2*)	✗	✗	✗	481MHz	8.05W (+5.10W)	10.35

Designs 7, 8, 9, 10, 11, 12, 15 have not been reported.
Comb: Combinatorial
*: DRC warning
**: DRC critical warning

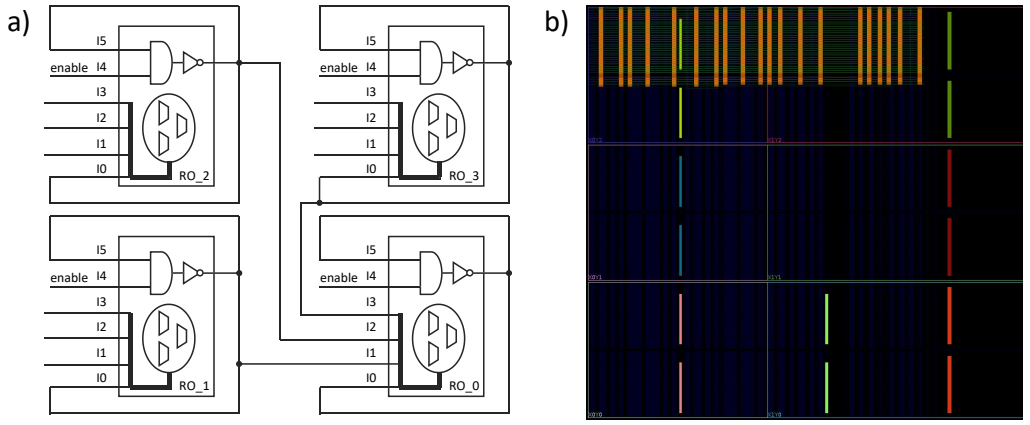


Fig. 10. Enhanced ROs grid for power-hammering: a) schematic; b) implementation with 2000 ROs.

inputs for implementing fastest possible ROs result in a wide range for both frequency and waste power (see Figure 12). We have discovered frequencies ranging from $1GHz$ to $6GHz$ approximately as a result of the internal architecture of the LUTs (see Figure 11) and a variance in the routing path delay for implementing the fastest possible loop. The corresponding waste power is not necessarily correlating with oscillator speed. Because we do not have access to the low-level ASIC details of any Xilinx FPGA, we cannot fully explain this behavior. Still, one possible explanation could be that longer paths are slower and have therefore a lower RO frequency; but because there are longer wires (with more capacitive load), more switching elements, and drivers involved per oscillator round-trip the overall waste power may still be high (or even higher).

Figure 11 shows the internal hierarchical architecture of a LUT, which is built from a tree structure of multiplexers where the inputs $I0$ to $I5$ are equal in their logical behavior but not for their timing. The figure shows that input $I0$ needs to travel through 6 levels of multiplexing to propagate to $O6$ which results in a primitive latency of $177ps$, while input $I5$ only needs to propagate one level resulting in $41ps$ latency. Moreover, because the adjacency of UltraScale+ switch matrices is relatively sparse (as usual for FPGAs), the fastest possible loop routing has a relatively high variance in latency depending on which specific LUT input is used for the loop. With this, we examined the fastest possible ROs where the loop routing can be implemented in just a single hop[1]. For these ROs, Vivado reported a path delay for the external routing ranging from $\approx 46ps$ to $\approx 71ps$. For having full control of the implementation throughout the experiments, we constraint the routing using the GoAhead tool [7]. Comparing the single-hop routing RO variants against each other is interesting because the variance in frequency is now mostly related to the internal latency inside the LUT itself (see Figure 15). The corresponding results are listed in Table 2.

3.3 Non-Combinatorial Self-Oscillator Variants

In addition to combinatorial loop-based ROs, non-combinatorial loops had been proposed in recent papers [19, 42, 57]. These designs use transparent latches, glitch amplification [19, 42, 57], or asynchronous reset/preset to create oscillators [19] (see also Section 4.3.2).

We repeated the experiments in [19, 42, 57] and confirmed that all designs could implement oscillators. Moreover, we manually optimized these oscillators for maximum speed by using different local routing options to fine-tune routing latencies.

[1]Here, a hop is actually passing two switch matrix multiplexers that together act as a pair to form a larger two-level multiplexer, similar as used for older Xilnix FPGA architectures (see also Figure 15).

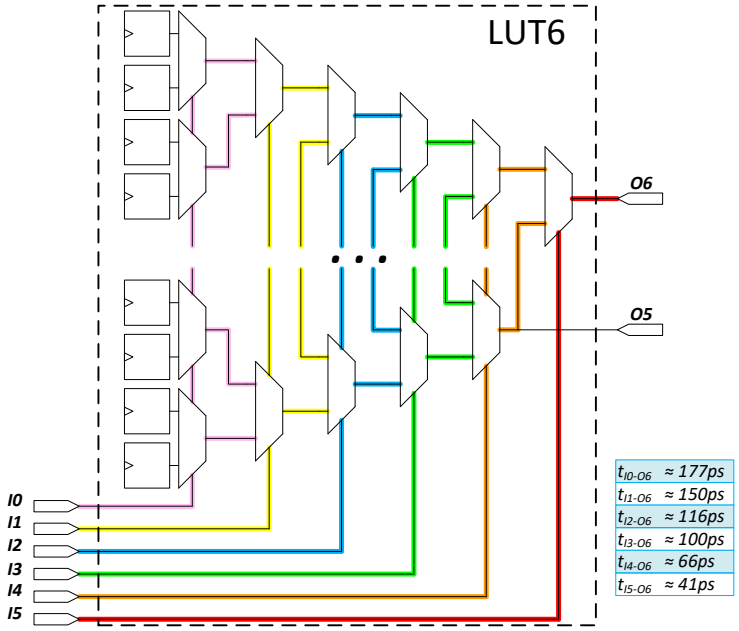


Fig. 11. Logical view of a Lookup Table 6-input primitive with timing information taken from Vivado.

In our experiment using glitch amplification (see design 15 in Table 2), we created glitches by creating routing paths with different signal propagation delays from a single T-flip-flop output to a LUT, which implements an XOR gate to create a glitch which is fed back to the clock input pin of the T-flip-flop. With a timing difference of 218ps between the two paths, we measured a frequency of 481MHz. It should be noted that this oscillator requires an external signal to kick-start the oscillator.

A common property shared among the here presented non-combinatorial loops is that they rely on local clock routing resources rather than on the global clock distribution network. We have not seen any use of clock routing resources for implementing the internal routing of High-Level Synthesis (HLS) generated circuits, and the clock distribution network is entirely used for clock signal routing. For the oscillator based on glitch amplification, the Vivado tool reported a gated clock (DRC warning code: `PDRC-153`) and a warning indicating a possible hold-time violation (DRC warning code: `PLHOLDVIO-2`).

3.4 Self-Oscillator Power Evaluation

So far, we reported timing characteristics of self-oscillating circuits and if the Xilinx vendor tools throw DRC error or warning messages that may or may not allow detecting oscillators in a design. In this section, we report our results on waste power that was drawn from the different oscillator designs, as shown in Table 2. From that table, we took the three most power-wasting designs (*Design 7, 8, 15*) to highlight their suitability for power-hammering attacks (see Table 3). To quantify the risk for power-hammering, we introduce the term Waste Power Potential (WPP), which we define as:

$$\begin{aligned} WPP &= \frac{possible\_waste\_power\_when\_using\_the\_whole\_FPGA}{total\_FPGA\_power\_budget} \\ &= \frac{PWP}{TP} \end{aligned} \tag{4}$$

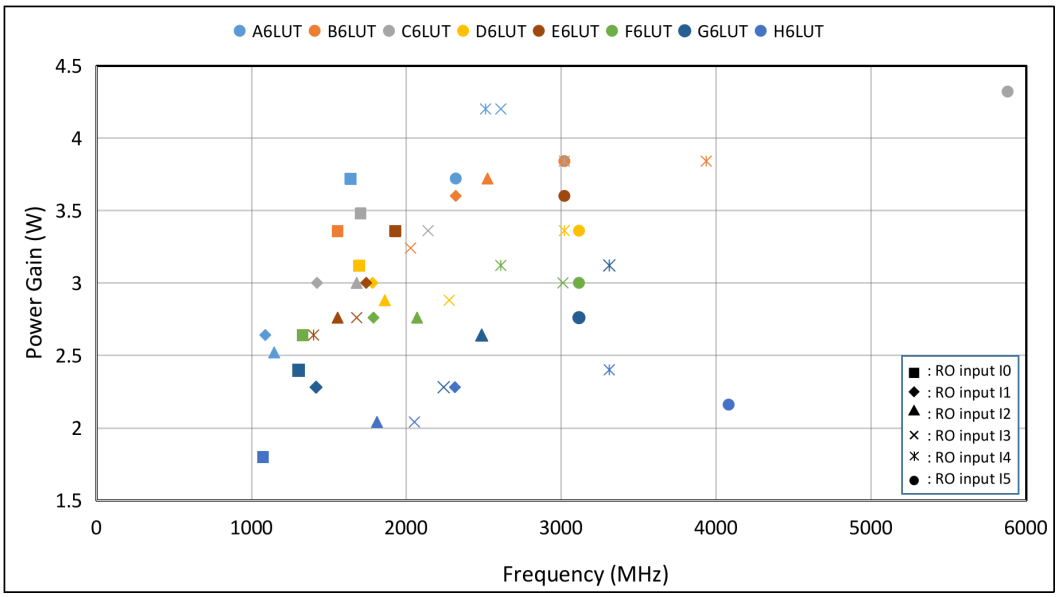


Fig. 12. ROs Frequency versus Waste Power Gain (measured for 2000 ROs) for all 8 LUT6 primitives inside a CLB for all corresponding different cases that implement the fastest possible loop from output O6 to an input of the same LUT (resulting in $8 \times 6 = 48$ individual experiments).

Where PWP (Possible Waste Power) denotes the assumed power consumed when a power-hammering circuit is occupying the entire FPGA and where TP (Total Power) refers to the power envelope typically defined by the power supply, the thermal design of the system, and the maximum power rating of individual components, including the FPGA. Depending on the system's power envelope, PWP may not be reachable in a particular system, and PWP is essentially expressing the potentially possible waste power. WPP reveals how a power wasting circuit performs per unit resources and unit power budget available in a particular system. $WPP < 1$ expresses that a power wasting circuit is likely not to be able to crash/harm the FPGA or system, while $WPP > 1$ expresses a potential risk to crash the FPGA. Moreover, the value of WPP denotes the number of resources needed to crash an FPGA. For example, with $WPP = 5$, an attacker can crash an FPGA by using at least 20% of the available resources. In reality, the threat will likely be even higher for power-hammering circuits that have a $WPP > 1$ because there will be other parts of the FPGA drawing some additional power (which could be incorporated by subtracting other power contributors from TP). Nevertheless, WPP is a good measure to quantify if a system is at risk of power-hammering. Please note that WPP assumes a steady waste power consumption and that even WPPs below one may cause harm due to dynamic voltage (IR) drops and other dynamic effects (e.g., resonance effects triggered in a power regulation circuit). However, the lower WPP, the lower the harm possible due to IR drops.

To maximize WPP, we amplified the power wasting effect caused by fast toggling ROs to additionally drive a large amount of local routing and logic elements for wasting even more power. Figure 10 shows the idea and implementation of our experiment. As shown, we intentionally connect each of the RO loops to some unused inputs of other ROs. These other ROs are placed in different CLBs to use more routing resources (e.g., wires, multiplexers, etc.) along the routing paths, which in turn wastes more power.

Figure 13 shows the power-hammering evaluation results on an Ultra96 board. $VCCINT$ is the core voltage of the FPGA which is recommended to be $0.85V$ [34]; VCC_SMPS is the voltage

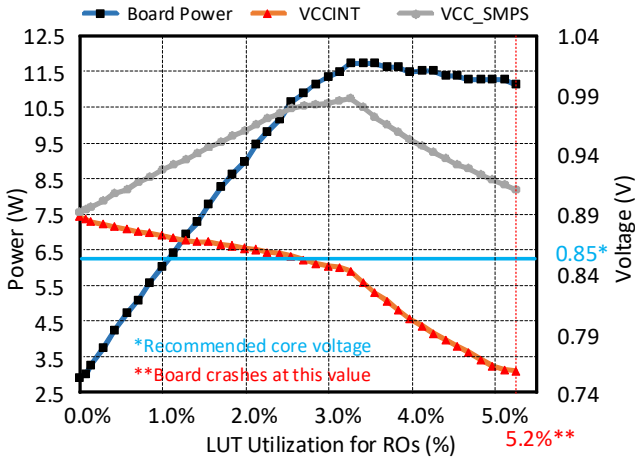


Fig. 13. Power-hammering Evaluation for Power over Core Voltage on Ultra96.

Table 3. Power-hammering evaluation between Xilinx Power Estimator, Measured Power Consumption on Ultra96, and Speculation Power Consumption on Alveo U200.

Designs from Table 2	LUT6 used	Power/Primitive	Xilinx Power Estimation	Power Gain in Ultra96	Provisioned Power Gain in Alveo U200 (with 50% LUTs utilization)	WPP
Design 7 - Dual-RO	2000	2.55 mW/Primitive	0.227W	5.10W	1507W	13.39
Design 8 - Enhanced ROs	2000	3.33 mW/Primitive	2.067W	6.66W	1968W	17.51
Design 15 - Glitch Amplification	6000	0.64 mW/Primitive	0.351W	5.10W	502W	4.47

measured at the output of the power supply regulator circuit for the FPGA; and *BoardPower* is measured at the $12V$ input to the Ultra96 board. From the result, we can see a gap between *VCC_SMPS* and *VCCINT* which relates to the voltage drop of the board's power supply network between the power supply regulator circuit to the FPGA. The increasing gap indicates a rise in the current until the power supply cannot compensate any longer and eventually crashing the board. We analyzed the schematic of the used Ultra96 board [5]. While the actual power regulator circuit and power drivers should be able to deliver over 10A to the FPGA, there is a TPS22920 load switch in the power network path which is rated for 4A and which has an on-resistance of $\approx 10m\Omega$ (at working temperature), which explains most of the *VCC_SMPS - VCCINT* gap.

Design 8 has a $WPP = 40.54$, and our experiments revealed that with only 6% of the available LUT resources (4000 LUTs of a ZU3EG FPGA), the used Ultra96 board crashed immediately. This number is higher than what is suggested by WPP where $1/WPP$ would be enough to impose a threat (which is $70k\ LUTs/40 \approx 1750 LUTs$ or 2.5% LUT resource for the used ZU3EG FPGA on an Ultra96 board). However, when studying Figure 13, we see that at $\approx$ 2.7% of the total LUT resources ($\approx$ 2000 LUTs), the core voltage starts to drop below the recommended core voltage, and the power supply starts to struggle to keep up with the demand resulting from the power-hammering. After that point, the power regulator circuit is unable to sustain the current demand resulting in a drop of *VCC_SMPS*. The tipping point when the core voltage drops below its nominal value matches quite close to the resources indicated by WPP.

The here presented results are even more significant when considering a datacenter FPGA card such as the Alveo U200 board from Xilinx. When assuming that our Zynq UltraScale+ power-hammering results can be directly transferred to the Xilinx Virtex UltraScale+ VU9P FPGA (because

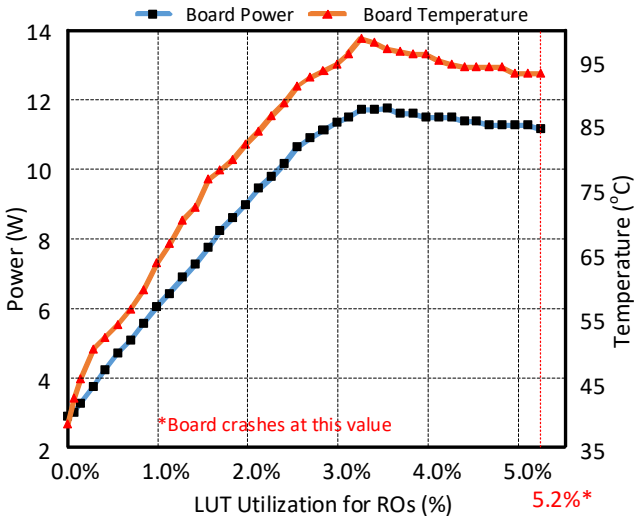


Fig. 14. Power-hammering Evaluation for Power over Temperature on Ultra96.

they use the exact same fabric architecture and the same 16nm FinFET process [31]), this would be equivalent to a WPP of 17.51 which translates into a total possible waste power of $PWP \approx$ **3940W**, when deploying *Design 8* on the entire VU9P of an Alveo U200 board (see also Table 2 and Table 3 for more results). This estimation is far beyond anything that the FPGA, the board, or the system would ever sustain, hence expressing the importance of preventing such circuits from getting configured on the FPGA in the first place.

Figure 14 shows the temperature of the board, corresponding to the number of deployed ROs Although cooling mechanisms (i.e. heatsinks, fans) keep the temperature below the maximum junction temperature, intensively heating the fabric may have a long term impact on the FPGA. This phenomenon is in particular dangerous if the heat is generated in a hot spot and not evenly spread across the entire FPGA die.

Additionally, we also implemented a "Dual-RO" (Figure 7a) exploiting the fact that a LUT6 primitive in UltraScale+ devices can be split into two individual LUT5 with shared inputs as shown in *Design 7* of Table 2. Thus, we can use both outputs of a fractural LUT to implement two independent oscillators. For 2000 LUTs (4000 ROs), which corresponds to less than 3% of the device capacity, we measured a waste power of $5.10W$. Please note that this is an increment in power, and the total power of the board was close to $8W$ and already close to the total power envelope of an Ultra96 board [4]. With this, the total possible waste power PWP is over $180W$, considering all the available 70K LUTs would be used for power-hammering.

4 FPGA VIRUS SCANNING AT THE ELECTRICAL LEVEL

4.1 Hardware versus Software Virus Scanning

In software systems, the confidentiality of data and task integrity are usually protected by different layers that may go beyond what a software virus scanner is testing. This includes protection mechanisms provided by the software operating system (OS) or run-time environments. Software binaries encode the functionality of a program primarily as sequences of instructions. Consequently, a virus scanner for software binaries will include a pattern matching engine, typically searching for regular expressions, which are also known as virus signatures.

Table 4. Contrasting protection mechanisms: software versus FPGA hardware techniques.

Software	Hardware
Check software binaries using regular expressions and cryptographic hash matches	Translate the FPGA bitstreams to netlists and check graph properties on netlists
Check address ranges to prevent executing malicious code	Check location information in configuration bitstreams
Bounds checking and memory randomization to prevent buffer overflows (eventually by an OS)	Check volume of configuration data written to prevent configuring adjacent regions

Contrarily, a configuration bitstream encodes the functionality of a module essentially as a netlist, which in turn is a structural representation given as configured FPGA primitives and configured switching elements (i.e. multiplexers). A netlist can be modeled as a graph, and the whole physical FPGA implementation process can be described by graph transformations, as summarized in 4.2 Consequently, *a virus scanner for FPGAs needs a checker engine for graph properties*. For example, in Section 3, we examined ring-oscillators in more detail that in one variant are implemented as cyclic combinatorial circuits (e.g., a LUT that has an output connected to its input without passing a flip-flop). By scanning a netlist (i.e. its corresponding graph representation), we can spot such oscillators.

A software OS and most software virus scanners commonly check memory addresses (or memory ranges) used by programs to ensure that data is not corrupted in malicious ways or that, for example, data segments are not executed as code. Similarly, a virus scanner for FPGAs has to check that a configuration bitstream is not corrupting the configuration context of other parts of the system, including other configurations or states of other parts of the system (e.g., the surrounding shell that commonly provides DDR memory and PCIe access). Consequently, a virus scanner needs a bitstream parser that ensures that a module will only change the configuration context of resources allocated to that module. This requirement includes parsing addressing information that encodes locations of primitives on the FPGA fabric as well as tracking the volume of configuration data that is written to the device. The latter tracking prevents a kind of a buffer overflow that can arise when configuring Xilinx FPGAs[2]. This attack would exploit that Xilinx FPGAs perform something similar to an auto-increment that keeps configuring an FPGA as long as it receives configuration data through a configuration port. As a consequence, this could overwrite the configuration of the fabric outside an intended module bounding box.

To some extent, the tracking of the configuration bitstream length is equivalent to buffer overflow detection and prevention techniques (e.g., bounds checking) as embedded into some compilers like the Clang frontend for LLVM [43].

We summarized the main differences between software and hardware virus scanning in Table 4. For full system security, it requires hardware support from the run-time system. For instance, systems commonly provide memory management units (MMUs) that can protect memory regions against malicious accesses. These units are also available in CPU-FPGA hybrids such as Xilinx Zynq UltraScale+ devices or Intel Stratix-10 SX SoC devices; and these chips include dedicated IOMMUs to protect the memory subsystem from malicious accesses initiated from the FPGA side (e.g., by an accelerator module). Alternatively, MMU functionality can be implemented in the FPGA's soft

[2]Please note that this problem is not necessarily bound to a specific vendor but that this problem is best understood for Xilinx FPGAs which dominate the research on run-time reconfigurable systems.

logic (commonly as part of a shell [66]). In this paper, we are, in particular, focusing on FPGA vulnerabilities at the electrical level because protecting a system at the system level is very well studied and, therefore, not further covered here.

4.2 Modelling the FPGA Virus Scanning Problem

Formally, any FPGA architecture can be modeled by its architecture graph $G_A = (V_A, E_A)$ which includes as its node primitives V_A^P and switches V_A^S with $V_A = V_A^P \cup V_A^S$ as well as directed edges E_A between the nodes representing wires or connections[3]. When a module is implemented for an FPGA, its specification (e.g., some RTL code) will undergo several transformation steps, including logic compilation, technology mapping, placement of primitives, routing, and ultimately the generation of the configuration bitstream. Concisely, we can say that the technology mapping is an allocation and mapping of primitive Boolean functions (the result of the logic synthesis step) to a set of connected primitives (including their internal configurations). The result of this step is a *netlist*, which is a graph $G_N = (V_N, E_N)$, where the nodes are FPGA primitives.

During placement, the nodes get placed on the architecture graph G_A, which is a binding β of the netlist nodes $V_N \rightarrow V_A^P$. In practice, this means that we annotate for each node in G_N the location coordinate L of the corresponding primitive of the FPGA:

$$V_N \rightarrow L(V_A^P), \forall\ V_A^P \in V_N$$

The process of routing can be defined as computing a binding of the netlist edges E_N to switches V_A^S and wires E_A. In general, this is a quite complicated process, and the actual routing has, among other things, to find spanning trees (for multiple edges $e_n \in E_N$ that have the same source node in V_N). Primitive nodes commonly have multiple input and output ports $p \in P_t$, where P_t is the set of ports for a specific primitive type t. The routing information can be seen as a set of switches (a list of nodes in V_A^S) and wires (a list of edges in E_A) that are used to implement each connection in E_N. The configuration of a switch is given by none (if the switch is not used) or exactly one edge from another node (or port in the case the source is a primitive node), which in turn represents the selected routing multiplexer input (e.g., in a switch matrix).

A placed and routed netlist can be directly mapped to a bitstream and encodes the exact configuration of each element $V_A^P, V_A^S \in V_A$. It is essential to understand that this mapping is *reversible*, meaning that a bitstream can be mapped back to a placed and routed netlist. However, this mapping needs the architecture graph to rebuild the routing, which is only encoded as segments in the bitstream, rather than as complete paths. On the contrary, a netlist generated through the implementation flow still provides a substantially higher level of abstraction than the bitstream. This is because a netlist typically includes information such as hierarchies, symbolic names of nets and logic blocks, and information on signal vectors, which cannot be easily decompiled from an FPGA configuration binary. This fact is similar to software compilation into obfuscated program binaries that also do not allow to decompile symbolic names and hierarchies.

This project uses the reversible correspondence between bitstream and netlist to rebuild flat netlists that provide all primitives V_N^P and all switching multiplexers V_N^S, but that will not offer any higher level information (such as symbol names or Boolean equations). For the remainder of this paper, we will use G_N to refer to a netlist that is rebuilt from a bitstream for detecting virus signatures.

Please note that the goal of this paper is not to provide/offer a reverse engineering tool for FPGAs but to show that configuration bitstreams are well suited to detect malicious circuit constructs in a module. Related work that focuses explicitly on reverse engineering includes [8, 61, 71].

[3]For the sake of clarity, we deliberately omit a discussion about bidirectional wires that had been available in older FPGA architectures.

4.3 Detecting Self-oscillating Circuits

As mentioned in Section 2, it is of paramount importance to identify self-oscillating circuits in a design to be deployed. The following paragraphs are devoted to different classes of self-oscillating designs.

4.3.1 Ring-Oscillators. Ring-Oscillators break the fundamental model of register-transfer level (RTL) descriptions where a circuit is described by

(1) registers, including FPGA slice flip-flops, pipeline registers (e.g., inside DSP primitives), or memories, and
(2) transforming logic that is forming *acyclic combinatorial paths*.

These paths can be described by a network of elementary Boolean functions implemented by look-up tables (LUTs), or DSP blocks[4] that are located between the registers.

It is a good design practice to follow the RTL design principle on FPGAs [62], and this is also the model commonly generated by High-Level Synthesis (HLS) tools [39]. In this paper, we assume that all states are stored in flip-flops or other synchronous memory elements (which is the typical case for FPGA designs). Circuit analysis using latches (which are sometimes used in ASIC netlists) is a well-studied topic, and there is no fundamental obstacle to transfer the here presented methodology to circuits based on latches.

To perform a search for cycles, we have to refine our netlist model so that each port $p \in P_t$ of a primitive V_N can be either a register P_t^R or a combinatorial element P_t^L for routing or logic. With this, we expand $\forall p \in P_t^R$ a path search that terminates at any other register port $\in P_t^R$ or that recursively explores all paths while keeping track for duplicate ports visited in P_t^L, which would indicate a cycle.

In general, it requires an *odd* number of inverters in a cycle to form a Ring-Oscillator. FPGADefender is not interpreting the logic blocks for possible inverters, and we deliberately scan for acyclic paths only. The reason for this is that if a combinatorial block (e.g., a LUT or DSP block) implements an inverter between an input and an output can depend on other inputs and consequently on a state that is only known when running a module. The philosophy of FPGADefender is to flag any possible oscillator while not report a false positive for any design that is following RTL design principles. Moreover, FPGADefender is stricter than the Xilinx vendor DRC checks, which can also detect some cycles, but there are situations where the vendor DRC fails. For example, an enabled transparent latch can be part of a Ring-Oscillator, and due to the latch (which is logically a wire), this cycle would not be flagged by the Xilinx vendor DRC but by FPGADefender. More examples are provided in Section 3.

For our implementation, we used Xilinx Vivado for generating a report file containing the full architecture graph for the used Zynq UltraScale+ XCZU3 FPGA. However, that model does not explicitly distinguish between P_t^R and P_t^L, and we added this annotation through a regular expression replacement. Furthermore, the path search inside the virus scanner incorporates all combinatorial primitives, including LUTs, DSPs, carry logic, cascading multiplexers (i.e. *F7Mux* and *F8Mux* in Xilinx nomenclature), which requires an understanding of the bitstream encoding of primitive control bits. However, the actual search does not have to distinguish between different types of primitives. This scan will identify the oscillator cases $1 \ldots 12$, which are reported in Section 3.

4.3.2 Self-clocking Oscillators. In our oscillator evaluation section (Section 3), we evaluated an oscillator circuit that is based on glitches that are generated by an XOR gate with different input

[4]For a sake of clarity, we are omitting a deeper discussion on pipeline registers in DSP blocks for the reminder of this paper, even FPGADefender can deal with all internal registers and pipeline stages in the DSP48 primitives which are available on Xilinx UltraScale+ FPGAs.

routing latencies and where the resulting glitches are fed back into the clock input of a toggle flip-flop for a self-propelled oscillation (see variant 15 in Table 3). In order to detect this kind of oscillator, we examine for each used flip-flop the corresponding clock source. If the source is a global clock network, the bitstream is accepted. If the clock source cannot be considered to be stable (e.g., a combinatorial LUT output), the bitstream is rejected. In our systems [17, 66], we block configuration access to global clock resources for any partially reconfigurable module by using BitMan [50] for preventing this kind of attack and FPGADefender will detect if partially reconfigurable modules try driving global clock resources.

4.3.3 Other Oscillators. In Section 3, we presented further self-oscillating designs that allow bypassing Vivado design rule checks (DRCs); therefore, this allows an attacker to implement oscillators which can be deployed in cloud data centers. To confirm this, we ran experiments on Amazon F1 instances for all Oscillator designs listed in Table 2 and all designs that don't throw any warning by the vendor tools can be deployed. The remainder of this paragraph will present the corresponding detection mechanisms.

The self-oscillator detection mechanism in Section 4.3.2 scanned for the origin of a clock source, and we use a very similar mechanism to handle asynchronous reset/preset inputs of slice flip-flops which can also be used for creating self-oscillating circuits (see variant 14 in Table 2). To detect this, we query the asynchronous mode flag from the bitstream for each used flip-flop, and in case any asynchronous reset/preset mode is used, we search from the control input (i.e. the reset/preset primitive input) backward to find the origin of the corresponding control signal. If the origin is a combinatorial primitive pin (P_t^L), the bitstream is rejected while we flag a warning for registers P_t^R.

In order to prevent the Vivado tool from flagging a combinatorial feedback loop, a transparent latch can be incorporated in the loop (see variant 13 in Table 2). We, therefore, treat latches as combinatorial elements (essentially like a wire) and carry out a loop search as described for ring-oscillators (Section 4.3.1). We also report a warning in the case latches are used.

The here presented tests allow detecting any FPGA implemented self-oscillating circuits that have been reported in the literature, including all further variants reported in Section 3.

4.4 Detecting Short-Circuits

In [2, 3, 6, 22], short-circuits had been reported that were implemented directly in the soft-logic on an FPGA. In older FPGA families (e.g., Xilinx Virtex-II), the fabric included some long-distance wires that could be accessed through tristate drivers at different positions, which could be used to create short-circuits inside the fabric. The deployable attack for short-circuits in modern FPGAs is based on the way switch matrix multiplexers are commonly implemented. In SRAM-based FPGAs, the multiplexers are implemented with transmission gates or pass-transistors [12] and by activating multiple inputs (i.e. switching on multiple transmission gates or pass-transistors within the same multiplexer), a short-circuit situation arises when the corresponding multiplexer inputs carry different logic levels. Therefore, by changing the input logic levels to the switch matrix multiplexers, it is possible to control the power that a shorted multiplexer is drawing precisely in time. This configuration provides the potential for denial-of-service-like (DoS-like) attacks.

As shown in Figure 15, 7-Series FPGAs from the vendor Xilinx implement a switch matrix multiplexer by cascading two levels of switching, each controlled through a one-hot coded configuration word (see [6, 44] for more details on FPGA switch matrix multiplexer implementations). In contrast, the multiplexers in UltraScale+ devices are smaller and use only one multiplexing level that is again one-hot encoded in the configuration bitstream. Consequently, for UltraScale+ devices, a used multiplexer input port corresponds directly to one specific configuration bit. Therefore, we

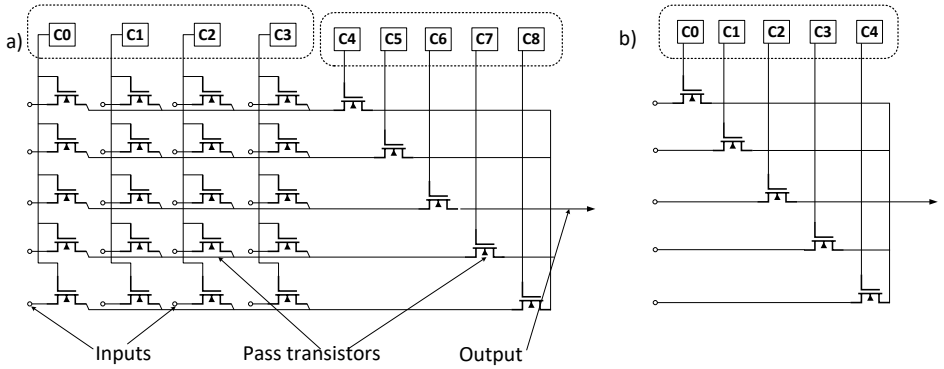


Fig. 15. a) Switch matrix multiplexer implementation on Xilinx 7-series FPGA; b) ditto for UltraScale+ FPGAs.

reject bitstreams where a switch matrix multiplexer encoding contains more than one bit among the set of bits that control that particular multiplexer.

Please note that the vast amount of UltraScale+ switch matrix multiplexers are used as pairs. Consequently, UltraScale+ multiplexers are similar to 7-Series multiplexers, with the main difference that internal multiplexer details are made visible to the user. This organization simplifies the short-circuit detection for UltraScale+ devices as it is not necessary to determine the sets of configuration bits that control a specific multiplexing level, as performed in [6] using graph algorithms (e.g., the configuration bits $C_0, \ldots, C_3$ in Figure 15a) form a set of configuration bits).

4.5 Bitstream Bounding-box Tests

Testing if a bitstream is exceeding its allocated (partial) region during configuration was examined in several projects before. For example, the configuration manager for the Erlangen Slot Machine project evaluated start address information and scanned the length of the bitstream written to the device [46]. The REPLICA project parsed configuration bitstreams directly in hardware as part of the configuration controller that connects to the configuration port of the FPGA [36]. A full overview of partial reconfiguration techniques is provided in [38]. For the virus scanning implemented in this paper, we use BitMan [50], which supports parsing of all Xilinx UltraScale+ FPGA configuration bitstreams. BitMan is used inside the FPGADefender flow for bounding box testing and for converting FPGA bitstreams to the required netlist for further graph search, as illustrated in Figure 1.

4.6 Detecting Wire-Tapping

The wire-tapping test checks if a partial module is connected to ports that belong to the static system or another module outside the circuit boundary (i.e. the region allocated to a reconfigurable module). However, a static signal may have to cross a reconfigurable region, like, for example, in order to access a gigabit transceiver (as part of the shell), and a module placed into this region should not be allowed to access this crossing signal. We define therefore prohibited ports/nodes in the architecture graph $p^- \in G_A$ (i.e. a *negative filter*) that are not allowed to exist in the netlist G_N which is corresponding to the circuit encoded by the bitstream to be examined: $p^- \notin G_N$. For convenience, we allow defining prohibited ports by regular expressions, which, for example, allows the definition of a bounding-box. This definition would be the same bounding-box as defined during system floorplanning when reconfigurable regions are defined (i.e. a P-block in Xilinx terminology). Because static routes may cross the area of a partial module differently in different systems, it is necessary to define the port list individually for each system. The port list for a static route can be

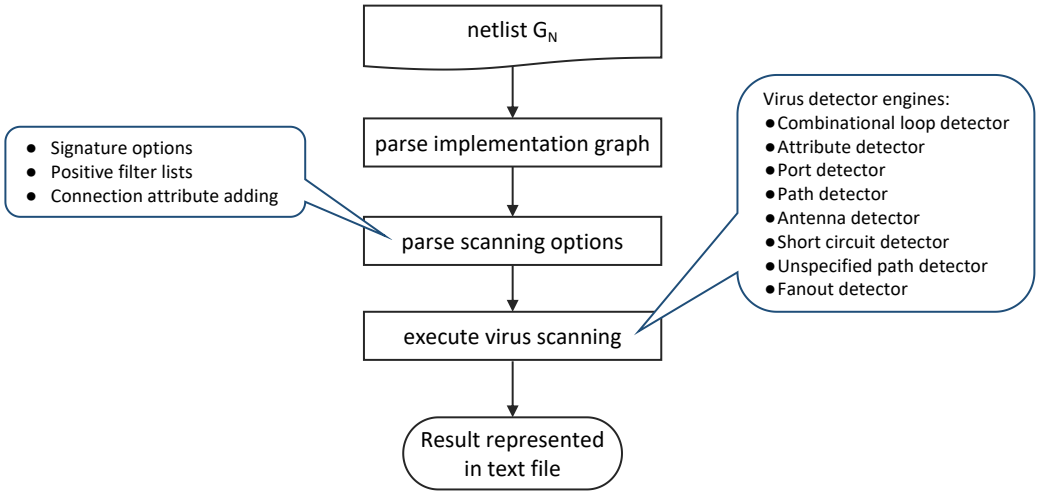


Fig. 16. FPGADefender flowchart.

easily derived automatically using the TCL interface in Vivado using the `get_property` command on the specific signals to be protected.

4.7 Interface Sanity Check

As mentioned in the previous section, we defined a *negative filter* for a set of ports ($p^- \in G_A$). FPGADefender can also search for port connections that must exist in a bitstream, which implements a *positive filter* ($p^+ \in G_N$). This filter is, in particular, used for partially reconfigurable modules to check if the module connects to the foreseen wires between the static system and the module such that no interface wires are left over as antennas. Only such interface wires implement the communication between a partial module and the surrounding while all other signals a strictly separated for both the surround (static) system and the partially reconfigurable module, as implemented in systems [51, 67]. An interface wire antenna may not necessarily indicate a malicious circuit but flags that a reconfigurable module may have an incompatible interface.

5 FPGADEFENDER: IMPLEMENTATION AND EVALUATION

This section is devoted to the implementation and evaluation of our FPGADefender bitstream virus scanner. The existing implementation was carried out for and tested on an Ultra96 board, but because FPGADefender operates on models that are automatically derived from the Xilinx Vivado tool suite, the approach is portable to all other Xilinx UltraScale+ FPGA platforms.

5.1 FPGADefender Implementation

```
{
  "begin": {"tile": {"name": "INT", "x": 18, "y": 19}, "name": "WW2_E_END6"},
  "end": {"tile": {"name": "INT", "x": 18, "y": 19}, "name": "
    INT_NODE_SDQ_34_INT_OUT0"},
  "attributes": []
},
```

Listing 1. A snippet of a single edge of a netlist graph.

FPGADefender[5] is built entirely in Python, which provides a bundle of supportive packages such as NetworkX [23] to represent and analyze netlist graphs derived from bitstreams using BitMan. BitMan, as discussed in Section 4, provides netlist graphs G_N which contain node information V_N and edge information E_N. The netlist graphs are encoded in `JSON` format, as shown in Listing 1.

After parsing a netlist graph G_N, scanning options are parsed to provide inputs for the virus detector engines as well as a set of positive filters $p^+ \in G_N$ and negative filters $p^- \in G_N$. Then the scanning process is executed based on a set of virus detector engines:

- **Combinational cycle detector:** Detect combinatorial cycles and transparent latch cycles.
- **Attribute detector:** Detect unusual synchronous design elements such as the use of latches.
- **Port detector:** Detect prohibited ports that are used in the bitstream.
- **Path detector:** Detect prohibited paths that are used in the bitstream.
- **Antenna detector:** Detect dangling paths in the bitstream (which indicate physical interface mismatches).
- **Short-circuit detector:** Detect short-circuits caused by possible bitstream manipulations (FPGADefender rejects bitstreams with invalid encodings for the routing).
- **Fanout detector:** Detect and report maximum fanout of the examined module.

A score is given in each of the scanning stages and summed up to deliver a total score. Based on the reported result, the configuration manager will be able to decide whether a bitstream is safe to be deployed or not, as shown in Figure 1. The specific security grading can be more complicated and depends on the security requirements of a specific system. For example, the FPGADefender scan result may be interpreted as strict or more relaxed (i.e. to allow latches). However, in any case, the scan result is quantifying precisely the risk potential of a partially reconfigurable configuration bitstream.

The actual implementations of the virus detector engines are based on set operator functions and graph traversals. This graph, however, considers FPGA-specific details in the FPGADefender implementation. For example, all modern FPGA architectures support LUTs that are fracturable. This fact means in the case of Xilinx FPGAs that a LUT6 can implement two independent LUT5 logic functions where the five lowest inputs are shared. For instance, we could think of a full adder where one LUT5 implements the sum result bit and the second LUT5 the carry to the next full adder. In Section 3, fractural LUTs were introduced for implementing 2 ROs in a single LUT.

In contrast to this, as shown in Figure 17, it is possible that a combinatorial path runs through one of the fracturable LUTs and later runs through the other fracturable LUT without forming a cycle or any RO. For such situations, it is not enough to analyze just the routing to decide if a netlist contains cycles or not. For example, in Section 4, the cycle scanning was introduced for single output LUTs. In the single output case, the path search is for each LUT input expanded to the output of the corresponding LUT. In the case of using the fractural LUT mode, expanding the search for each LUT input to both LUT outputs could result in flagging false positives (i.e. flagging cycles in a netlist that cannot implement ROs). This result would happen in the example shown in Figure17 for `LUT_A` when, for example, the most bottom LUT input would expand a path search to the top LUT output despite that the bottom LUT input can only affect the bottom LUT output. We solved this problem by analyzing the LUT function table (i.e. the LUT init values in the bitstream) using the Espresso logic minimizer [11] with the help of the PyEDA library for Python [15]. By optimizing the LUT table values to provide a logic optimized Boolean expression, we know which of the LUT inputs affect each of the individual LUT outputs of a fracturable LUT. We, therefore, only expand the path search for detecting cycles for LUT inputs to a LUT output when the input affects that LUT output. With this, we circumvent the false positive problem for fracturable LUTs.

[5] Available online at `https://github.com/KasparMatas/FPGAVirusScanner`.

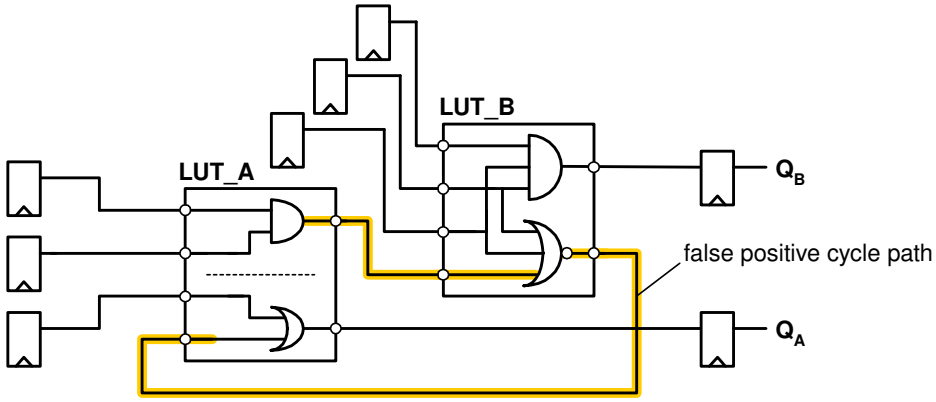


Fig. 17. Example of a path that closes in LUT_A but that does not form a cycle or RO.

5.2 Design Evaluation

In order to test and evaluate FPGADefender, we developed several test cases including 1) 15 malicious designs from Table 2 as well as a short-circuit design; and 2) 28 reference designs including the Spector OpenCL benchmark [16], soft-core CPUs (MIPS and RISC-V [70]), crypto cores (AES, DES [64], and SHA3 [49]) and other peripheral circuits [48], which all do not contain any malicious circuits (see Table 5 for the list of malicious circuits and Table 6 for list of all normal test cases).

FPGADefender found all malicious circuits and the short-circuits in our test cases. For the experiments, each malicious circuit from *Design 1 to 15* was implemented 2000 times spread out across the FPGA. The *Short-Circuit* design was created directly at the bitstream level. This was implemented with the help of BitMan, which features a low-level API to access LUT values and switch matrix multiplexer configurations. To inject short-circuits, we looked for valid one-hot encoded switch matrix multiplexer configurations and randomly toggled some of the zero bits (to create randomly more hots in the multiplexer configurations). Note that the Vivado design tool does not allow to create a bitstream that contains any short-circuit. This restriction means that short-circuits can be prevented if, for example, a cloud service provider generates the bitstream at the provider side rather than accepting a bitstream binary from a user. However, this also implies that users have to share their design with the cloud service provider (at least to some extent through design checkpoints - DCPs), which in turn is an IP protection issue. In contrast to this, FPGADefender would allow a cloud service provider to directly accept FPGA configuration bitstreams while being able to detect short-circuits.

It should be noted that the FPGA vendor Xilinx does currently not provide any tool or mechanism to scan a bitstream for any malicious construct or any manipulation. The only requirement FPGADefender imposes to perform its virus scanning is a plain (i.e. non-encrypted) bitstream.

So far, we tested each threat in isolation. For more rigid testing, we created designs that mix different threats and tested each time if FPGADefender finds all threats correctly just by scanning the configuration bitstream. In these experiments, FPGADefender correctly flagged all threats in all test cases.

We compared the FPGADefender combinatorial cycle detection with the Xilinx DRC checker (See Table 2). Here, FPGADefender did not only detected correctly *Design 1 to 8* but also detected the hidden combinatorial cycles from MUX primitives in *Design 9 and 10*, CLA primitive in *Design 11*, and even cycles through the DSP primitive, as in *Design 12*, where the Xilinx tool fails.

Table 5. Evaluation results for malicious designs circuits.

Designs	LUT used	Comb Cycle	Latch	Short-Circuit	Fanout
Malicious Circuits					
Design 1	2000	2000	0	No	1
Design 2	2000	2000	0	No	1
Design 3	2000	2000	0	No	1
Design 4	2000	2000	0	No	1
Design 5	2000	2000	0	No	1
Design 6	2000	2000	0	No	1
Design 7	2000	4000	0	No	2
Design 8	2000	2000	0	No	10
Design 9	0	2000	0	No	1
Design 10	0	2000	0	No	1
Design 11	0	2000	0	No	1
Design 12	0	2880	0	No	1
Design 13	2000	2000	2000	No	1
Design 14 *	4000	0	0	No	3
Design 15 *	6000	0	0	No	4
Short Circuit	15974	11669	12	Yes	5223

* Designs are flagged as flip-flop clock input violation.

In *Design 13*, ROs are implemented through transparent latches. While Xilinx failed to flag those ROs (even if the latch enable is activated by a constant), FPGADefender found all cyclic paths that run through latches.

In *Design 14 and 15*, combinatorial logic paths are used to drive the clock input of flip-flops instead of global clock sources. This will be flagged using the path detector engine in FPGADefender.

As a sanity check, we used FPGADefender to scan all the 28 bitstreams of the test cases that are not intentionally designed with malicious constructs. FPGADefender has not detected malicious constructs except for one case, the true random number generator (TRNG). The TRNG uses ring-oscillators as a source of randomness, which all got flagged by FPGADefender. This case is a dilemma that could be solved by providing primitives by a system vendor (e.g., a cloud service provider) for exceptional use cases like TRNGs or PUFs (Physical Unclonable Functions). For the *True Random Number Generator* using ROs, we found exact 128 combinatorial cycles corresponding to the 128-bit random number generated.

6 DISCUSSION AND CONCLUSIONS

In this paper, we provided a complete investigation of self-oscillator threats deployable on FPGAs. We systematically researched all published and several new oscillator designs for implementing such self-oscillating circuits, considering all logic, arithmetic and memory primitives on a Zynq UltraScale+ FPGA from the vendor Xilinx and we considered a range of different modes of operation to create self-oscillation circuits (e.g., combinatorial loops, glitch amplification, and asynchronous flip-flop modes of operation). In particular, we examined and quantified the threat potential for

power-hammering, and we found for an optimized design that by using just 3% of the LUT resources of an Ultra96 board, we can draw more power than the allocated power budget. This result means that an attacker would only need control over a small amount of the total FPGA resources to crash an FPGA board or even cause permanent damage to a system. By giving strong evidence about the risk potential using small FPGAs for the experiments, we can project results onto new data center FPGAs that use the same FPGA fabric architecture and manufacturing process technology. Furthermore, several of the here researched oscillator designs are deployable on FPGA cloud service instances, as we tested for Amazon F1 instances with potential power-hammering potentials in the range of kilowatts, which has enormous potential for harming equipment with a corresponding substantial monetary risk.

Due to similarities in how different FPGA architectures of different FPGA vendors are physically implemented, the here presented attack vectors are not bound to a specific vendor. However, different vendors provide different details on their devices and tools, and the more information is released, the more defenses can be implemented. For instance, Intel has not released a full FPGA architecture graph or any details on the bitstream encoding for their recent FPGAs. Therefore, anybody who wants to perform an independent security assessment of a bitstream has to carry out a substantial amount of reverse engineering work. The same holds for adding support for Intel FPGAs in FPGADefender. *We believe that security through obscurity is a bad practice and that only fully documented devices should be considered for building secure systems.*

We strongly believe that a security-first strategy is imperative for existing and future FPGA ecosystems and that business models based on end-users having access to FPGA hardware can only be carried out with an FPGA hardware security infrastructure in place.

We addressed this issue with the development of FPGADefender which not only can identify (probably) any kind of self-oscillating circuit but in addition to essential threats like short-circuits in the interconnect and tapping of wires outside of the scope of a user module. With this, FPGADefender can prevent all recent approaches for remote side-channel analysis [18, 54, 55] and all popular power hammering attacks [20]. Future work will investigate if potential malicious power-hammering from glitches (e.g., from XOR trees) can be detected reliably at the bitstream level.

We strongly believe that all FPGA hardware security issues can be tackled by tools, and we see no fundamental obstacle that would prevent building systems allowing user access to an FPGA, including multi-tenancy and the direct deployment of bitstreams.

In order to make this happen and to stimulate more research on hardware security, FPGADefender, as well as a gallery with design checkpoints and bitstreams of malicious circuits, is provided under: `https://github.com/KasparMatas/FPGAVirusScanner`.

ACKNOWLEDGMENT

This work is kindly supported by the UK National Cyber Security Centre through the project *rFAS* (grant agreement 4212204/RFA 15971) and by the European Commission through the project *EuroEXA* (grants 754337). We also thank the Xilinx University Program for providing tools and boards.

Table 6. Evaluation results for benchmarking circuits.

Designs	LUT used	Comb Cycle	Latch	Short-Circuit	Fanout
Normal Circuits					
8b10b EncDec *	72	0	0	No	15
CAN Controller *	1310	0	0	No	146
BCD Adder *	68	0	0	No	6
PRNG *	237	0	0	No	107
Cordic *	1312	0	0	No	99
I2C *	307	0	0	No	87
Parallel Scrambler *	66	0	0	No	11
RS232 UART *	102	0	0	No	19
SPI *	988	0	0	No	174
Stepper Motor *	69	0	0	No	9
Breadth First Search (BFS) †	604	0	0	No	204
DCT †	10085	0	16	No	418
FIR Filter †	3842	0	4	No	749
Histogram †	2409	0	0	No	217
Merge Sort †	2905	0	1	No	235
Matrix Multiplication †	8116	0	9	No	1782
Normal Estimation †	8504	0	6	No	620
Sobel Filter †	14045	0	0	No	272
SPMV †	10670	0	9	No	1552
Black-Scholes ‡	12326	0	10	No	259
RISC-V CPU ‡	3556	0	0	No	170
AES §	4520	0	0	No	162
DES §	278	0	0	No	20
Mandelbrot §	1716	0	42	No	183
MIPS CPU §	4163	0	0	No	572
SHA3 §	10662	0	0	No	262
Skin Color Detection §	2022	0	0	No	147
TRNG §	1069	128	0	No	61

* Peripheral IP designs from OpenCores [48].
† Open-source OpenCL designs from Spector benchmark [16].
‡ Other open source designs [45, 70].
§ Academic handcrafted RTL designs [49, 64].

REFERENCES

[1] A. C. Aldaya, A. Sarmiento, and S. Sánchez-Solano. 2016-04. AES T-Box Tampering Attack. *Journal of Cryptographic Engineering* 6, 1 (2016-04), 31,48.

[2] R. Amerson, R. Carter, W. Culbertson, P. Kuekes, G. Snider, and L. Albertson. 1996. Plasma: An FPGA for Million Gate Systems. In *Fourth International ACM Symposium on Field-Programmable Gate Arrays*. 10–16.

[3] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes, and G. Snider. 1995. Teramac-Configurable Custom Computing. In *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*. 32–38.

[4] Avnet. 2018. Ultra96 Hardware User Guide. Retrieved Oct 29, 2019 from http://zedboard.org/sites/default/files/documentations/Ultra96-HW-User-Guide-rev-1-0-V0_9_preliminary.pdf

[5] Avnet. 2018. Ultra96 Schematics. Retrieved Oct 29, 2019 from https://github.com/96boards/documentation/blob/master/consumer/ultra96/ultra96-v1/hardware-docs/files/ultra96-schematics.pdf

[6] C. Beckhoff, D. Koch, and J. Torresen. 2010-08. Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration. In *2010 International Conference on Field Programmable Logic and Applications*. IEEE, 596,601.

[7] C. Beckhoff, D. Koch, and J. Torresen. 2012. Go Ahead: A Partial Reconfiguration Framework. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 37–44.

[8] F. Benz, A. Seffrin, and S. A. Huss. 2012. Bil: A tool-chain for bitstream reverse-engineering. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 735–738.

[9] E. Biham and A. Shamir. 1997. Differential Fault Analysis of Secret Key Cryptosystems. In *Annual international cryptology conference*. Springer, 513–525.

[10] A. Bradbury, L. James, L. Marques, T. Roberts, P. Vogel, P. Wagner, and S. Elliott. 2019. LowRISC -Running on the FPGA. Retrieved Sep 15, 2019 from https://www.lowrisc.org/docs/debug-v0.3/fpga/

[11] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Vol. 2. Springer Science & Business Media.

[12] C. Chiasson and V. Betz. 2013. Should FPGAs Abandon the Pass-gate?. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–8.

[13] Intel Corp. 2018. White Paper: Secure Device Manager for Intel ®Stratix ®10 Devices Provides FPGA and SoC Security. Retrieved Oct 28, 2019 from https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01252-secure-device-manager-for-fpga-soc-security.pdf

[14] J. Danger, S. Guilley, S. Bhasin, and M. Nassar. 2009. Overview of Dual Rail with Precharge Logic Styles to Thwart Implementation-level Attacks on Hardware Cryptoprocessors. In *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*. 1–8.

[15] C. Drake. 2018. Python Electronic Design Automation. Retrieved Oct 29, 2019 from https://pyeda.readthedocs.io/en/latest/2llm.html

[16] Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. 2016. Spector: An OpenCL FPGA Benchmark Suite. In *FPT*.

[17] K. Georgopoulos, K. Bakanov, I. Mavroidis, I. Papaefstathiou, A. Ioannou, P. Malakonakis, K. D. Pham, D. Koch, and L. Lavagno. 2019. *A Novel Framework for Utilising Multi-FPGAs in HPC Systems*. 153–189.

[18] I. Giechaskiel, K. Rasmussen, and K. Eguro. 2016. Leaky Wires: Information Leakage and Covert Communication Between FPGA Long Wires. (2016), 15–27.

[19] I. Giechaskiel, K. Rasmussen, and J. Szefer. 2019. Measuring Long Wire Leakage with Ring Oscillators in Cloud FPGAs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*.

[20] D. Gnad, F. Oboril, and M. Tahoori. 2017. Voltage Drop-based Fault Attacks on FPGAs Using Valid Bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–7.

[21] T. Güneysu and A. Moradi. 2011. Generic Side-Channel Countermeasures for Reconfigurable Devices. In *Cryptographic Hardware and Embedded Systems – CHES 2011*, Bart Preneel and Tsuyoshi Takagi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 33–48.

[22] I. Hadžić, S. Udani, and J. Smith. 1999. FPGA Viruses. In *International Workshop on Field Programmable Logic and Applications*. Springer, 291–300.

[23] A. Hagberg, P. Swart, and D. Schult. 2014. NetworkX - Software for Complex Networks. Retrieved Oct 29, 2019 from https://networkx.github.io/

[24] Amazon Inc. 2019. Amazon EC2 F1 Instances. Retrieved Jun 27, 2019 from https://aws.amazon.com/ec2/instance-types/f1/

[25] Alibaba Inc. 2019. Deep Dive into Alibaba Cloud F3 FPGA as a Service Instances. Retrieved Jun 27, 2019 from https://www.alibabacloud.com/blog/deep-dive-into-alibaba-cloud-f3-fpga-as-a-service-instances_594057

[26] Amazon Inc. 2020. AWS FPGA: Programmer's View of the Custom Logic. Retrieved Mar 07, 2020 from https://github.com/aws/aws-fpga/blob/master/hdk/docs/Programmer_View.md

[27] Baidu Inc. 2020. FPGA Cloud Server. Retrieved Mar 05, 2020 from https://cloud.baidu.com/product/fpga.html

[28] Nimbix Inc. 2020. Xilinx Alveo Accelerator Cards. Retrieved Mar 05, 2020 from https://www.nimbix.net/alveo

[29] Xilinx Inc. 2018. Using Encryption and Authentication to Secure an UltraScale/UltraScale+ FPGA Bitstream. Retrieved Oct 28, 2019 from https://www.xilinx.com/support/documentation/application_notes/xapp1267-encryp-efuse-program.pdf
[30] Xilinx Inc. 2018. Vivado 2018.02. https://www.xilinx.com/products/design-tools/vivado.html
[31] Xilinx Inc. 2019. Delivering a Generation Ahead at 20nm and 16nm. Retrieved Oct 29, 2019 from https://www.xilinx.com/about/generation-ahead-16nm.html
[32] Xilinx Inc. 2019. UltraScale Architecture DSP Slice. Retrieved Oct 29, 2019 from https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf
[33] Xilinx Inc. 2019. UltraScale Architecture Memory Resources. Retrieved Oct 29, 2019 from https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf
[34] Xilinx Inc. 2019. Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics. Retrieved July 19, 2019 from https://www.xilinx.com/support/documentation/data_sheets/ds925-zynq-ultrascale-plus.pdf
[35] Y. Ishai, A. Sahai, and D. Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology - CRYPTO 2003*, Dan Boneh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–481.
[36] H. Kalte, G. Lee, M. Porrmann, and U. RÃ¼ckert. 2005. REPLICA: a Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In *19th IEEE International Parallel and Distributed Processing Symposium.*
[37] N. Kamoun, L. Bossuet, and A. Ghazel. 2009. Correlated Power Noise Generator as a Low Cost DPA Countermeasures to Secure Hardware AES Cipher. In *2009 3rd International Conference on Signals, Circuits and Systems (SCS).* 1–6.
[38] Dirk Koch. 2012. *Partial Reconfiguration on FPGAs: Architectures, Tools and Applications.* Vol. 153. Springer Science & Business Media.
[39] D. Koch, F. Hannig, and D. Ziener. 2016. *FPGAs for Software Programmers* (1st ed.). Springer Publishing Company, Incorporated.
[40] J. Krautter, D. Gnad, F. Schellenberg, A. Moradi, and M. Tahoori. 2019. Active Fences against Voltage-based Side Channels in Multi-Tenant FPGAs. Retrieved Oct 29, 2019 from https://eprint.iacr.org/2019/1152.pdf
[41] J. Krautter, D. Gnad, and M. Tahoori. 2018. FPGAhammer: Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2018), 44–68.
[42] J. Krautter, D. Gnad, and M. Tahoori. 2019. Mitigating Electrical-level Attacks Towards Secure Multi-Tenant FPGAs in the Cloud. *ACM Trans. Reconfigurable Technol. Syst.* 12, 3, Article 12 (Aug. 2019), 26 pages.
[43] C. Lattner. 2019. Clang: a C Language Family Frontend for LLVM. Retrieved Jun 25, 2019 from https://clang.llvm.org/
[44] D. Lewis, E. Ahmed, G. Baeckler, V. Betz, M. Bourgeault, D. Cashman, D. Galloway, M. Hutton, C. Lane, A. Lee, P. Leventis, S. Marquardt, C. McClintock, K. Padalia, B. Pedersen, G. Powell, B. Ratchev, S. Reddy, J. Schleicher, K. Stevens, R. Yuan, R. Cliff, and J. Rose. 2005. The Stratix II Logic and Routing Architecture. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays.* ACM, New York, NY, USA, 14–20.
[45] L. Ma, F. B. Muslim, and L. Lavagno. 2016. High Performance and Low Power Monte Carlo Methods to Option Pricing Models via High Level Design and Synthesis. In *EMS.* 157–162.
[46] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. 2007. The Erlangen Slot Machine: A Dynamically Reconfigurable FPGA-based Computer. *J. VLSI Signal Process. Syst.* 47, 1 (April 2007), 15–31.
[47] S. S. Mirzargar and M. Stojilovic. 2019. Physical Side-Channel Attacks and Covert Communication on FPGAs: A Survey. In *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications (FPL).*
[48] OpenCores. 2020. Free and Open Source gateware IP cores. Retrieved Feb 27, 2020 from https://opencores.org/
[49] K. Pham, E. Horta, D. Koch, A. Vaishnav, and T. Kuhn. 2018. IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs. In *2018 IEEE 12th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC).* 36–43.
[50] K. D. Pham, E. Horta, and D. Koch. 2017. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.* IEEE, 894–897.
[51] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch. 2018. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *Proceedings of the 5th International Workshop on FPGAs for Software Programmers (FSP).*
[52] G. Provelengios, D. Holcomb, and R. Tessier. 2019. Characterizing Power Distribution Attacks in Multi-User FPGA Environments. In *Proceedings of the 29th International Conference on Field-Programmable Logic and Applications (FPL).*
[53] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Xiao, and D. Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14).* IEEE Press, Piscataway, NJ, USA, 13–24.
[54] C. Ramesh, S. Patil, S. Dhanuskodi, G. Provelengios, S. Pillement, D. Holcomb, and R. Tessier. 2018. FPGA Side Channel Attacks without Physical Access. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* IEEE, 45–52.

[55] F. Schellenberg, D. Gnad, A. Moradi, and M. Tahoori. 2018. An Inside Job: Remote Power analysis Attacks on FPGAs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1111–1116.
[56] F. Schellenberg, D. R. E. Gnad, A. Moradi, and M. B. Tahoori. 2018. Remote Inter-Chip Power Analysis Side-Channel Attacks at Board-Level. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–7.
[57] T. Sugawara, K. Sakiyama, S. Nashimoto, D. Suzuki, and T. Nagatsuka. 2019. Oscillator without a Combinatorial Loop and Its Threat to FPGA in Data Centre. *Electronics Letters* 55, 11 (2019), 640–642.
[58] P. Swierczynski, G. Becker, A. Moradi, and C. Paar. 2018-03-01. Bitstream Fault Injections (BiFI)-Automated Fault Attacks Against SRAM-Based FPGAs. *IEEE Trans. Comput.* 67, 3 (2018-03-01), 348,360.
[59] P. Swierczynski, M. Fyrbiak, P.p Koppe, A. Moradi, and C. Paar. 2017. Interdiction in practice—Hardware Trojan against a high-security USB flash drive. *Journal of Cryptographic Engineering* 7, 3 (01 Sep 2017), 199–211.
[60] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar. 2015-08. FPGA Trojans Through Detecting and Weakening of Cryptographic Primitives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 8 (2015-08), 1236–1249.
[61] SymbiFlow. 2019. Project X-Ray. Retrieved Sep 15, 2019 from https://github.com/SymbiFlow/prjxray
[62] V. Taraate. 2019. *Advanced HDL Synthesis and SOC Prototyping*. Springer US.
[63] S. Trimberger and J. Moore. 2014-08. FPGA Security: Motivations, Features, and Applications. *Proc. IEEE* 102, 8 (2014-08), 1248,1265.
[64] A. Vaishnav, J. R. G. Ordaz, and D. Koch. 2017. A Security Library for FPGA Interlays. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–4.
[65] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside. 2018. Resource Elastic Virtualization for FPGAs Using OpenCL. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 111–1117.
[66] A. Vaishnav, K. D. Pham, K. Manev, and D. Koch. 2019. The FOS (FPGA Operating System) Demo. Retrieved Sep 15, 2019 from https://github.com/khoapham/fos
[67] M. Vesper, D. Koch, and K. Pham. 2017. PCIeHLS: an OpenCL HLS framework. In *FSP 2017; Fourth International Workshop on FPGAs for Software Programmers*. 1–6.
[68] R. Watanabe, S. Ura, Q. Zhao, and T. Yoshida. 2019. Implementation of FPGA Building Platform As a Cloud Service. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2019)*. ACM, New York, NY, USA, Article 6, 6 pages.
[69] A. Wild, A. Moradi, and T. Güneysu. 2018. GliFreD: Glitch-Free Duplication Towards Power-Equalized Circuits on FPGAs. *IEEE Trans. Comput.* 67, 3 (March 2018), 375–387.
[70] Clifford Wolf. 2019. PicoRV32. Retrieved Feb 27, 2020 from https://github.com/cliffordwolf/picorv32
[71] T. Zhang, J. Wang, S. Guo, and Z. Chen. 2019. A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code. *IEEE Access* 7 (2019), 38379–38389.
[72] M. Zhao and G. Suh. 2018. FPGA-based Remote Power Side-channel Attacks. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 229–244.
[73] K. Zick and J. Hayes. 2012-03-01. Low-cost Sensing with Ring Oscillator Arrays for Healthier Reconfigurable Systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 5, 1 (2012-03-01), 1,26.
[74] K. Zick, M. Srivastav, W. Zhang, and M. French. 2013. Sensing Nanosecond-scale Voltage Attacks and Natural Transients in FPGAs. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 101–104.