Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond

Kaspar Matas, Tuan La, Nikola Grunchevski, Khoa Pham, and Dirk Koch
Department of Computer Science,
The University of Manchester, UK
{firstname.lastname}@manchester.ac.uk

ABSTRACT

Since FPGAs are now available in datacenters to accelerate applications, providing FPGA hardware security is a high priority. FPGA security is becoming more serious with the transition to FPGA-as-a-Service where users can upload their own bitstreams. Full control over FPGA hardware through the bitstream enables attacks to weaken an FPGA-based system. These include physically damaging the FPGA equipment and leaking of sensitive information such as the secret keys of crypto algorithms. While there is no known attacks in the commercial settings so far, it is not so much a question of *if* but more of *when*? The tutorial will show concrete attacks applicable on datacenter FPGAs.

The goal of this tutorial is to prepare the FPGA community to impending security issues in order to pave way for a proactive security. First, we will give a tour through the FPGA hardware security jungle surveying practical attacks and potential threats. We will reinforce this with live demos of denial of service attacks. Less than 10% of the logic resources on an FPGA can draw enough dynamic power to crash a datacenter FPGA card. In the second part of the tutorial, we will show different mitigations that are either vendor supported or proposed by the academic community. In summary, the tutorial will communicate that while FPGA hardware security is complicated to bring about, there are acceptable solutions for known FPGA security problems.

KEYWORDS

FPGA security, hardware security, datacenter, cloud computing

ACM Reference Format:
Kaspar Matas, Tuan La, Nikola Grunchevski, Khoa Pham, and Dirk Koch. 2020. Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '20), February 23–25, 2020, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3373087.3375390

1 INTRODUCTION

Traditionally, FPGA industry had the position that hardware security of an FPGA was primary about protecting designs in terms of intellectual property (IP) in configuration data (i.e. the configuration bitstream) against cloning/overbuilding, reverse engineering, tampering, and spoofing, as summarized in [24]. This view has changed with FPGAs are now being integrated into data centers and cloud computing infrastructures at large scale [4, 10, 19]. One principle commonly used in cloud computing is *resource pooling* which allows sharing resources across different tenants such that overall utilization of cloud hardware resources gets improved. Resource pooling is currently not offered by any major FPGA cloud service provider, but multi-tenant scenarios are expected to provide better utilization and consequently better overall power efficiency at lower cost as compared to the current one-user-per-fabric scheme [25]. It should also be mentioned that the commonly used scenario consisting of a shell (i.e., the static system infrastructure that a data center FPGA provides to allow a user circuit communicating with the server) and the user accelerator design can already be considered multi-tenant. This is because the shell and a tenant are implemented individually and it needs protection mechanisms to ensure the system integrity of both (shell and user accelerator). For instance, a user accelerator should not be able to gain access to the shell, which in turn may compromise other parts of the cloud infrastructure.

The tutorial continues as follows: the next section provides a survey on FPGA hardware security, followed in Section 3 with a tutorial on how to research optimized ring oscillators for power hammering and for side-channel attacks. This section serves as a template to create other kinds of malicious circuits. Section 4 provides a tutorial on installing and using the open-source FPGA bitstream virus scanner FPGADefender. Some virus scan results are finally provided in Section 5.

2 HARDWARE THREATS FOR DATACENTER FPGAS

Due to their deep low-level programmability, FPGAs comprise new threat models far beyond of 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 [5, 21]. These kind of attacks are not available in known software threat models, but had been shown for FPGAs.

In the reminder of 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 user confidentiality (via physical side-channel analysis)

This section will also provide published state-of-the-art countermeasures.

FPGA '20, February 23–25, 2020, Seaside, CA, USA

ACM ISBN 978-1-4503-7099-8/20/02.
https://doi.org/10.1145/3373087.3375390

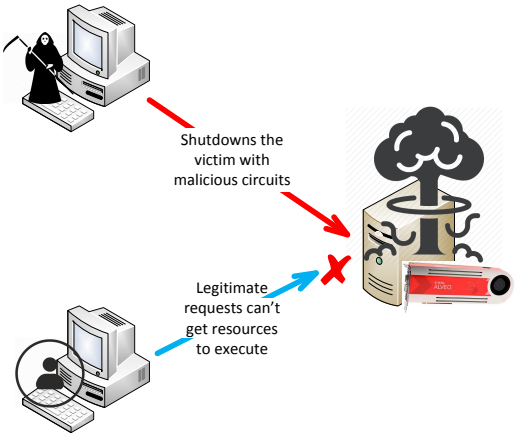

Figure 1: Denial-of-service-like (DoS-like) threat model. A user may try to shutdown an FPGA service in a data center by sending malicious circuits such that legitimate requests from other users cannot use the FPGA resources. Short-circuits and power hammering designs can be utilized for such attacks on the system availability. This kind of attack may potentially age or damage the equipment.

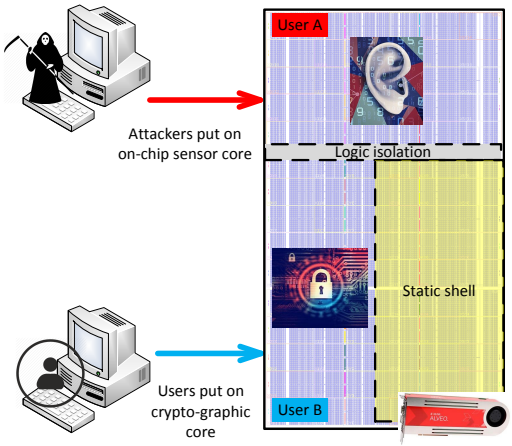

Figure 2: An illustration of the *eavesdropping* threat model of user confidentiality in a multi-tenant computing environment.

2.1 Attacks on the system availability

Denial-of-service-like (DoS-like) attacks are used to bring down active infrastructures and/or to compromise states in other system components which stay outside the scope of an attacking module, as illustrated in Figure 1. At the electrical level, two means for DoS-like attacks had been utilized: short-circuits and power hammering.

Short-circuits on modern FPGAs have been demonstrated in [1] within the multiplexers inside a switch matrix using a manipulated configuration bitstream resulting in a huge current increase (with several mA extra current for a single multiplexer). While the FPGA vendor tools ensure that generated bitstream are short-circuit free, an attacker can create shorts relatively easily. In fact, in [8], short-circuits had been used for obfuscating power traces from an AES core to make power analysis attacks much harder to perform.

Power hammering is another mechanism to carry out DoS-like attacks. All current power hammering attacks [7] are based on fast toggling circuits in order to draw a substantial amount of dynamic power. We will show in Section 3 that it is possible to implement ring oscillators running in the GHz frequency domain with a corresponding dynamic power footprint. In [7], a grid of ring oscillators was 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). In this tutorial, we will examine the potential for power hammering in more detail in Section 4. Although ring-oscillators are usually flagged with a warning by the vendor design tool flows and hence, not allowed to be deployed on any cloud or data center infrastructure, a recent research [6] has reported new ring-oscillator designs which can bypass such Design Rule Checking (DRC). A simple trick to bypass DRC is implementing a ring oscillator passing through an enabled transparent latch. With this, ROs could be deployed, for example, on Amazon F1 instances.

Although all current power hammering attacks are leveraging self-oscilating circuits, glitch amplification can potentially be utilized for this purpose as shown in Figure 3.

2.2 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 [16]). In [14], timing faults have been injected through a large number of ring oscillators to cause voltage drops followed by analyzing the resulting faulty cipher text using Differential Fault Analysis (DFA) for successfully revealing the secret key of a crypto-core. The idea of most timing fault injection attacks is to temporarily create a huge power demand (e.g., by starting a large number of ring oscillators). This will reduce the FPGAs supply voltage and may in turn slow down a path in a victim circuit such that it may fail timing.

Power analysis attacks have been demonstrated to leak the secret key of a cryptographic function that was running on the same FPGA [21], running on a CPU embedded on the same FPGA die [27], and running on a different FPGA on the same FPGA board [22]. 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 crosstalk effects [5, 6, 20]. In these studies, it was found that a long wire carrying a logical 1 will slow down a ring-oscillator that is implemented using an adjacent 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.

2.3 State-of-the-art countermeasures

The main schemes to prevent side-channel power analysis attacks are based on *masking* and *hiding* strategies. In the masking strategy, an implementation of a cryptographic algorithm is transformed to another (typically larger) variant which is functionally equivalent, but where the new circuit is able to remain secure although the attacker can observe some details of the operation through a side-channel, as proposed in [11]. This makes power analysis attacks much harder as the data leaked has also to be correlated with the

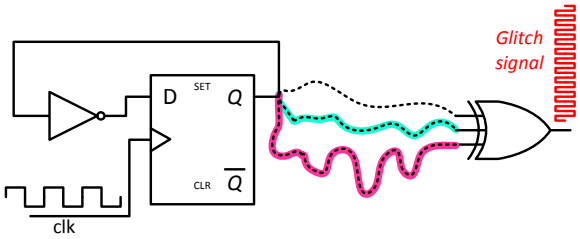

Figure 3: Illustration of a glitch generator. The glitch signal can be used to drive a large network of wires and combinatorial logic to drain excessive power.

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, as suggested in [3, 8, 12, 26].

Ring-oscillators can be used to monitor the healthiness of an FPGA fabric [28] and can also detect voltage drop attacks (e.g., power hammering, power analysis) [18, 29]. A recent work has suggested to use ring-oscillators not only to monitor a power analysis attack but also to response against the attack by triggering more power noise [13].

In a recent related work [15], LUT-based ring-oscillator designs are detected directly from configuration bitstreams. While that work fundamentally showed that oscillator circuits can be detected from bitstreams, it was only shown for basic LUT-based oscillators. This leaves an attacker the chance to deploy alternative oscillator designs (e.g., based on glitch amplification). Furthermore [15] 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 that are substantially more complex (e.g., fracturable LUTs, complex DSP blocks with ALU functionality, complex clock networks, a hierarchical routing fabric, etc.).

The following section will show how we use GoAhead to research the threat potential of ring-oscillators, while in Section 4 we show how viruses[1] can be detected with our new tool FPGADefender.

3 OPTIMIZING AND EVALUATING RING OSCILLATOR DESIGNS FOR POWER HAMMERING AND SPEED

For power hammering, an attacker obviously wants to maximize the amount of power a malicious circuit can waste per unit resources. For a side-channel attack respectively, the highest sensitivity is the most important objective, which correlates to the speed of an oscillator. In this section, we will use the tool GoAhead to design and tune ring oscillators to waste as much power as possible or to run as fast as possible. We use a setup on an Ultra96 Board where we precisely measured supply power. We use a Time-to-Digital Converter, as illustrated in Figure 4, to measure the actual clock frequency of an oscillator. The FPGA on the Ultra96 board uses the same manufacturing process and the same UltraScale+ FPGA fabric architecture than what is provided in current Xilinx datacenter FPGA boards, like the popular Alveo U200/250 FPGA boards. The best design found for Ultra96 will then be tested at scale on an Alveo U200 board.

[1]This tutorial uses the term *virus scanning* in its figurative meaning for detecting all kinds of malicious threats rather than in its original meaning of infecting a program with malicious code to spread out in a virus-like manner.

3.1 Time-to-Digital Converters on Xilinx UltraScale FPGAs

Ring oscillators on an FPGA can run in the GHz regime which is substantially faster than any user logic design can normally sustain. This makes the use of simple counters prohibitive to measure fastest possible oscillators. We therefore used a Time-to-Digital Converter (TDC) to measure speeds of oscillators. TDCs basically use propagation delay to measure a wave form. The idea of TDCs is to use different propagation delays from the probe to a set of flip-flops such that the parallel sampled flip-flops reveal the state of the probe at different points in time. See Figure 4 for an illustration on the operation of a TDC.

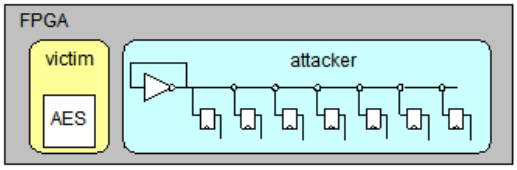

Figure 4: Illustration of a Time-to-Digital Converter (TDC). The speed of the ring oscillator is measured by a chain of flip-flops that sample a delay line in parallel.

The flip-flop sample clock speed can be selected mostly arbitrary (we used 100 MHz) as the variance in propagation delay is the key property that determines the characteristics of a TDC. Traditionally, TDCs had been implemented using carry chains to implement the delay chain. However, Xilinx UltraScale+ FPGAs do not have traditional carry chains, but use carry-look-ahead (CLA) circuits instead. For implementing a TDC on Xilinx UltraScale+ FPGAs, we consequently use local routing for the delay chain. Using this strategy, it is not important to find the fastest path (which we are normally interested in when implementing a module for performance), but instead finding paths that reach the different flops such that we form a TDC delay chain that has a linear characteristic (i.e. the variance in time between any pair of two consecutive flip-flops should be about the same) and high resolution (i.e. the absolute delay between any pair of two consecutive flip-flops should be small). This physical implementation problem is non-trivial and not directly supported by the Xilinx vendor tools.

We solved that problem using the path search function in the tool GoAhead [2]. GoAhead is a tool originally designed for implementing partially reconfigurable FPGAs. In its latest version, it uses Xilinx Vivado to report a device description that includes the entire architecture graph (including all possible switch matrix settings) as well as a detailed timing model. This device description is parsed in by GoAhead which allows this tool to report latencies for any path found from any arbitrary primitive port or a port inside a switch matrix. To automate processes in GoAhead, the tool supports TCL scripts. The TCL script in Figure 5 provides an example for searching for all possible paths stating from `startPort` on tile

`startTile` to reach a set of flip-flops specified in `flopPins` located on `targetTile`.

```
# ------ SETUP ------

set startTile CLEL_R_X0Y1
set startPort CLE_CLE_L_SITE_0_DMUX

set targetTile CLEL_R_X0Y0
set prefix CLE_CLE_L_SITE_0_
set flopPins [list AQ AQ2 BQ BQ2 CQ CQ2
DQ DQ2 EQ EQ2 FQ FQ2 GQ GQ2 HQ HQ2]

# ------ Functionality ------

for {set i 0} {$i < [llength $flopPins]}
  {incr i}
  { set targetPort ${prefix}
                [lindex $flopPins $i]

    PathSearchOnFPGA
        SearchMode=BFS
        Forward=True
        KeepPathsIndependet=False
        BlockUsedPorts=False
        OutputMode=CHAIN
        StartLocation=$startTile
        StartPort=$startPort
        TargetLocation=$targetTile
        TargetPort=$targetPort
        MaxSolutions=5
        MaxDepth=10
        PrintBanner=True
        # latencies in the order SLOW_MIN,
        # SLOW_MAX, FAST_MIN, FAST_MAX)
        PrintLatency=true,false,false,false
        FileName=OutputFile.txt
        Append=True
        CreateBackupFile=True;
}
```

Figure 5: GoAhead TCL script example to find paths from one primitive pin to a set of flip-flops.

The results of the script is a set of paths found by a breadth-first search sorted for each path in the `for-loop` of the script by the number of hops (which correlates to the routing resources used), as shown in Figure 6. Please note that the names used in GoAhead correspond to exactly the same naming scheme used by Xilinx in their Vivado tool suite. This holds for names used in scripts as well as for names used in results. Most importantly, GoAhead annotates the latency for each path. As can be seen in Figure 6, GoAhead reports the time as it gets incremented along the path. With the `PrintLatency` switch in the GoAhead `PathSearchOnFPGA` function, a user can select between any `SLOW_MIN`, `SLOW_MAX`, `FAST_MIN`, or `FAST_MAX` timing corner to be considered in the timing analysis.

For building the TDC delay chain, the result paths are sorted by their latency (i.e. the latency reported for the last hop in each path)

```
-----------------------------------------
-- From CLEL_R_X0Y1.CLE_CLE_L_SITE_0_DMUX
-- to CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AQ
-----------------------------------------
CLEL_R_X0Y1.CLE_CLE_L_SITE_0_DMUX (Latency: 0) ->
INT_X0Y1.LOGIC_OUTS_E5 (Latency: 0) ->
INT_X0Y1.INT_NODE_SDQ_1_INT_OUT1 (Latency: 0.004) ->
INT_X0Y1.SS1_E_BEG1 (Latency: 0.045) ->
INT_X0Y0.SS1_E_END1 (Latency: 0.045) ->
INT_X0Y0.INODE_E_9_FT1 (Latency: 0.059) ->
INT_X0Y0.BOUNCE_E_0_FT1 (Latency: 0.072) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AX (Latency: 0.072) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AQ (Latency: 0.072)

CLEL_R_X0Y1.CLE_CLE_L_SITE_0_DMUX (Latency: 0.072) ->
INT_X0Y1.LOGIC_OUTS_E5 (Latency: 0.072) ->
INT_X0Y1.INT_NODE_SDQ_1_INT_OUT1 (Latency: 0.076) ->
INT_X0Y1.SS1_E_BEG1 (Latency: 0.117) ->
INT_X0Y0.SS1_E_END1 (Latency: 0.117) ->
INT_X0Y0.INT_NODE_IMUX_30_INT_OUT1 (Latency: 0.132) ->
INT_X0Y0.BYPASS_E1 (Latency: 0.149) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_BX (Latency: 0.149) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AQ (Latency: 0.149)

...

-----------------------------------------
-- From CLEL_R_X0Y1.CLE_CLE_L_SITE_0_DMUX
-- to CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AQ2
-----------------------------------------
CLEL_R_X0Y1.CLE_CLE_L_SITE_0_DMUX (Latency: 0) ->
INT_X0Y1.LOGIC_OUTS_E5 (Latency: 0) ->
INT_X0Y1.INT_NODE_SDQ_1_INT_OUT1 (Latency: 0.004) ->
INT_X0Y1.SS1_E_BEG1 (Latency: 0.045) ->
INT_X0Y0.SS1_E_END1 (Latency: 0.045) ->
INT_X0Y0.INODE_E_9_FT1 (Latency: 0.059) ->
INT_X0Y0.BOUNCE_E_0_FT1 (Latency: 0.072) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AX (Latency: 0.072) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_AQ2 (Latency: 0.072)

...
```

Figure 6: Output created by the TCL script in Figure 5.

and we manually chose a set of paths that result in good linearity ($\approx \pm 10ps$) and a reasonably fine resolution ($\approx 70ps$). With N_{HIGH} samples in a TDC for the measured high values and N_{LOW} being the number of low samples, the speed of a RO is:

$$f_{RO} \approx \frac{1}{t_{delay} \times (N_{HIGH} + N_{LOW})} \quad (1)$$

And with a $\approx 70ps$ resolution, this allows measuring RO speeds up to about 7GHz[2]. In order to get even more accurate resolution, we are reporting all values as the median of at least 10 000 runs.

3.2 Optimizing Ring Oscillators for Power Hammering and Speed

With having an FPGA system instrumented for accurate measurement of power and frequency measurement (through our TDCs), we explored and evaluated various FPGA ring-oscillator designs for

[2]We would like to highlight that a relatively cheap FPGA board allows experimenting at such high speeds.

```
-----------------------------------------
-- From CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A_O
-- to CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A_O
-----------------------------------------
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A_O (Latency: 0) ->
INT_X0Y0.LOGIC_OUTS_E0 (Latency: 0) ->
INT_X0Y0.INT_NODE_IMUX_0_INT_OUT0 (Latency: 0.021) ->
INT_X0Y0.IMUX_E22 (Latency: 0.042) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A4 (Latency: 0.042) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A_O (Latency: 0.142)

CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A_O (Latency: 0.142) ->
INT_X0Y0.LOGIC_OUTS_E0 (Latency: 0.142) ->
INT_X0Y0.INT_NODE_IMUX_0_INT_OUT0 (Latency: 0.163) ->
INT_X0Y0.BYPASS_E3 (Latency: 0.189) ->
INT_X0Y0.INT_NODE_IMUX_10_INT_OUT1 (Latency: 0.214) ->
INT_X0Y0.IMUX_E26 (Latency: 0.233) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A5 (Latency: 0.233) ->
CLEL_R_X0Y0.CLE_CLE_L_SITE_0_A_O (Latency: 0.297)

...
```

Figure 7: Output created by a GoAhead path search using the same port for begin and target (for finding ring oscillator variants).

their suitability for power hammering and for side-channel attacks (i.e., fastest oscillator speed). We used the GoAhead tool to find all possible ring oscillator designs. This uses again the GoAhead `PathSearchOnFPGA` command (which we used for designing the TDC in the previous section) by simply specifying the output of the LUT intended for the ring oscillator implementation for both the `beginPort` and the `targetPort`. An output of such a search is shown in Figure 7. For each LUT, the path search will sort the result paths found in an order reporting the paths with the least number of hops first. These paths are typically the fastest ones and the reported latency serves as a sanity check. We used a GoAhead script (in the same way used for finding the TDC delay path) to find all fastest ring-oscillator designs over all LUTs in a CLB. We then implemented those paths for 2000 LUTs on the Ultra96 board and measured speed and power consumption.

Our experiments found the fastest oscillator speed being 5.8GHz and an increase in power of 4.2W for the most malicious oscillator design found (see Figure 9). The experiments with the poorest results achieved only 1.1GHz speed and 1.7W waste power. This means that a single LUT has a waste power potential of 2.1mW when considering the most malicious oscillator design.

To put this into perspective: an Alveo U200 data center card featuring a VU9P FPGA providing 1.182 million LUTs would have a waste power potential of over 2kW using the optimized power ring oscillator design. Consequently even a fraction of that logic would by far exceed the thermal and electrical specifications of any FPGA/FPGA board.

3.3 Xilinx Alveo U200 Power Hammering Experiment

We deployed the optimized ring-oscillator design from the previous paragraph on an Alveo U200 data center card. This board has the same specifications than the FPGA boards available with Amazon's F1 instances. We deployed 384000 ROs (≈ 32% of the available LUTs,

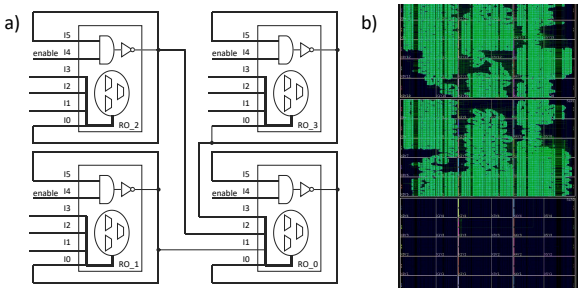

Figure 8: Enhanced ROs grid for power hammering: a) schematic; b) implementation with 384000 ROs.

as shown in Figure 8) and gradually enabled them to evaluate the critical point when the board crashes. Surprisingly, when reaching only 15% of the total LUTs resources (1182240 LUT6 primitives), it causes a strong drop in internal core voltage *VCCINT* and eventually crashing the board when *VCCINT* reached 0.74*V*. At that point, we already exceeded the 225W maximum power budged of the board. Figure 10 and Figure 11 show the power consumption, internal FPGA core voltage *VCCINT*, and core temperature in relation to the activated ROs. Please note that the power was measured on the power supply grid with the help of an Ampere meter (*True RMS*). The used power supply was a *Silverstone Strider 600W Modular SFX 80+ Gold Power Supply*. The figures illustrate how dangerous malicious circuits could be in a data center setup. Therefore, it is necessary to prevent loading any bitstream onto an FPGA board that may include such malicious circuits.

3.4 Further GoAhead Use Cases

So far, we showed how the timing-driven path search in GoAhead can be used to find and optimize ring-oscillators. There are several other use cases that can benefit from this ability, in particular in the field of hardware security. For instance, in Figure 3 we showed how different routing latencies can cause glitches. This however, depends on the exact routing delays and by balancing latencies for all paths to the XOR gate shown in the example in Figure 3, glitches can actually be canceled out. This is relevant for implementing DPA-resistant circuits of cryptographic algorithms that often heavily use XORs. In such applications, balancing routing latencies may dramatically reduce power signatures that can be measured by a potential attacker (see also Figure 2).

Vice versa, carefully imbalanced routing can be used for amplifying glitches (as needed for power-hammering). Other use cases include the design of asynchronous circuits and wave pipelining that rely on the implementation of exact (routing) latencies to function correctly.

In many cases, only a few signals are critical and they can be easily found by a path search in GoAhead together with a ranking of the results by latency. The paths selected can be directly implemented in the Xilinx Vivado tool through guided routing constraints (using the TCL command `set property ROUTE`). All remaining routing can then be added by Vivado automatically. By default, GoAhead uses a breadth-first search which means that the search essentially enumerates the entire search space. In practice, this is often acceptable because the depth of the search is rather limited (typically less than 10 hops in practical systems) and the

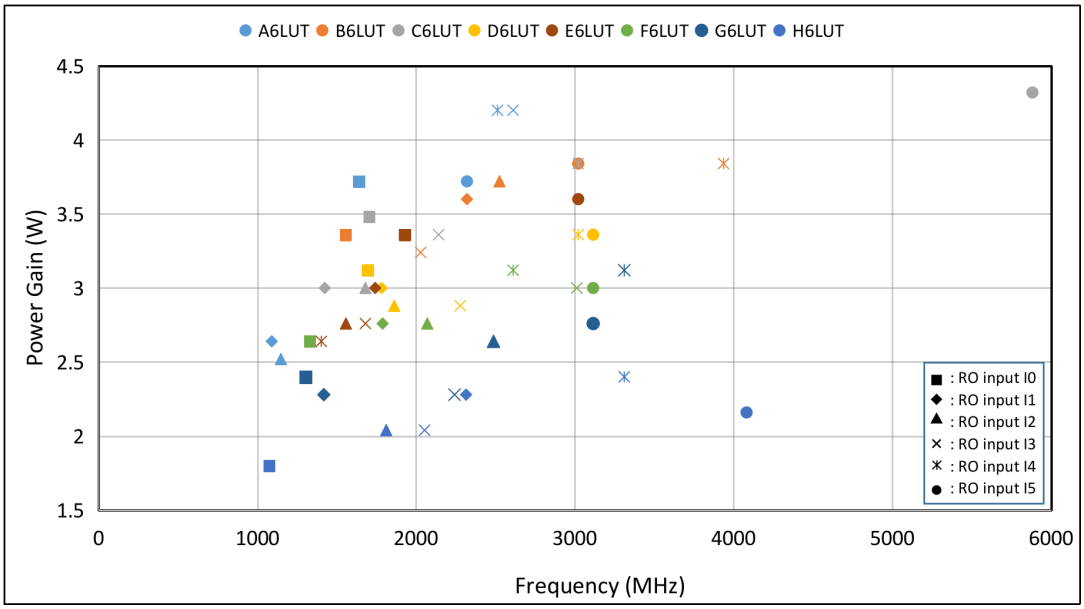

Figure 9: ROs Frequency versus Waste Power Gain (measured for 2000 ROs) for all 8 LUT6 primitives inside a CLB for all corresponding different cases that implement the fastest possible loop from output O6 to an input of the same LUT (resulting in $8 \times 6 = 48$ individual experiments).

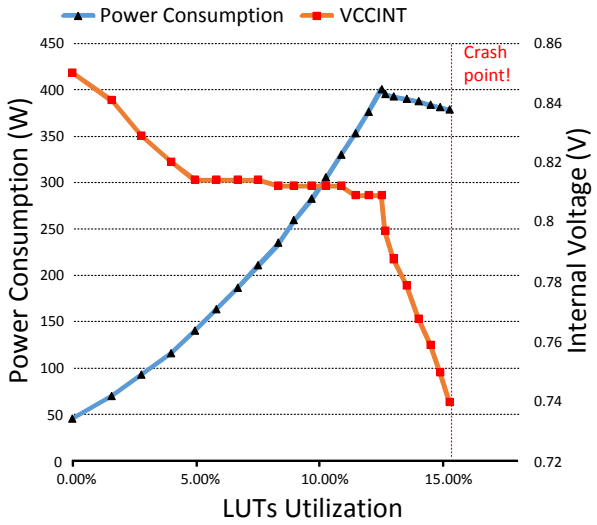

Figure 10: Power Consumption and Core Voltage versus LUTs Utilization on Alveo U200.

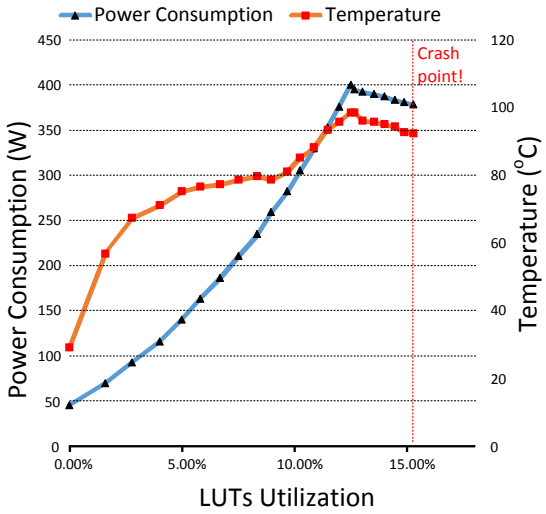

Figure 11: Power Consumption and Temperature versus LUTs Utilization on Alveo U200.

adjacency of switch matrices is rather sparse. For longer paths, the GoAhead path search also supports a variant of A*.

4 FPGA VIRUS-SCANNING WITH FPGADEFENDER

Having examined the threat of ring-oscillators in previous sections, we are now looking closer into threat mitigation strategies. We will now introduce the tool FPGADefender which detects malicious constructs in bitstreams such that a system can reject a threat before it could even materialize on an FPGA.

4.1 Overview

FPGADefender[3] is built entirely in Python which provides a bundle of supportive packages such as NetworkX [9] to represent and analyze an implementation graph from a bitstream. As a first step, an implementation graph is created by a netlist generator which contains node and edge information. This graph reassembles the netlist encoded inside the bitstream. The netlist generator is implemented as an enhancement to the tool Bitman[4]. The implementation graph is encoded in JSON format as shown in Figure 13.

[3]Available online at: https://github.com/KasparMatas/FPGAVirusScanner.git
[4]Bitman [17] is available under: https://github.com/khoapham/bitman.git

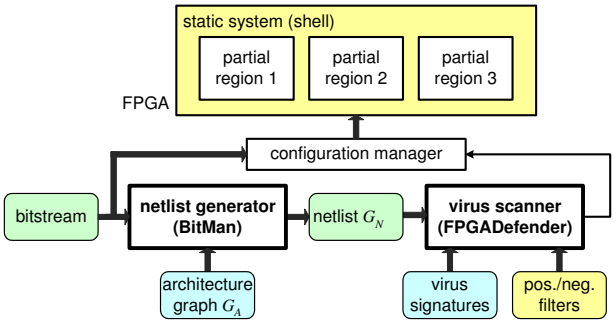

Figure 12: Envisioned system with a virus scanner for detecting malicious configuration bitstreams.

```
[
  {
  "begin": {
    "tile": {"name": "DSP", "x": 18, "y": 160},
    "name": "DSP_DSP48_1_ACOUT_B24"},
  "end": {
    "tile": {"name": "DSP", "x": 18, "y": 165},
    "name": "DSP_DSP48_0_ACIN_B24_PIN"},
  "attributes": []
  },

  {
  "begin": {
    "tile": {"name": "INT", "x": 3, "y": 74},
    "name": "INT_NODE_IMUX_1_INT_OUT0"},
  "end": {
    "tile": {"name": "INT", "x": 3, "y": 74},
    "name": "IMUX_E4"},
  "attributes": []
  },
  ...
]
```

Figure 13: A snippet of a single edge of the implementation graph.

After parsing the implementation graph, scanning options are parsed to provide inputs for the virus detector engine as well as filters. FPGADefender allows specifying a *positive filter* to describe configurations that must exist in the original bitstream (e.g., a specific connection through which a partially reconfigurable module communicates with the surrounding shell infrastructure). Correspondingly, a *negative filter* allows describing primitives and routing resources that are prohibited in a bitstream. In detail, the scanning process executes the following set of virus detector engines:

- Combinational cycle detector: Detect combinatorial cycles. This includes detecting cycles that use transparent latches in order to prevent the attack revealed in [23].
- Attribute detector: Detect asynchronous design elements such as using latches.
- Port detector: Detect prohibited ports. For example, this allows it to detect if a partial module tries leaking to a port not belonging to its allocated partial region.
- Path detector: Detect prohibited paths. For example, detect if a partial module tries accessing a static route that is crossing a partial region (note that we explicitly allow static routes which are commonly used in complex designs).
- Antenna detector: Detect dangling paths. This is in most cases rather a warning that a module may have an interface wire not properly connected.
- Short circuit detector: Detect short circuits caused by bitstream manipulation. In general, we detect any bitstream encoding that is invalid for routing. In Xilinx UltraScale+ FPGAs, this means in practice that all switch matrix multiplexers have to be one-hot encoded.
- Fanout detector: Detect and report maximum fanout. This is an indicator for a malicious design as power-hammering needs some kind of high fan-out control in order to activate a larger number of ROs. However, this is just an indicator as an attacker could easily hide high fan-out signals. This is an interesting field for further research.

A score is given in each scanning stage and summed up to deliver a total score. Currently, FPGADefender is leaving the evaluation of the scores and the report to the user. However, our virus scanner performs already all the heavy-lifting scanning work. Based on the reported result, the configuration manager will ultimately be able to decide whether a bitstream is safe to be deployed or not, as shown in Figure 12.

4.2 How to use FPGADefender

FPGADefender is a command-line program for scanning implemented FPGA designs (i.e. bitstreams) for malicious circuits and constructs. This section will describe installing and using FPGADefender as well as showing some scan examples. All of the examples below use the short option flags. For more details about the options use the `--help` flag.

Given design in file `input_design.json`, a scan can be performed on the command line using:

```
virusscanner -i input_design.json -c config.
    ini -o output.txt
```

The above command runs FPGADefender on the implemented graph given by the `input_design.json` file based on the options set in the `config.ini` file and outputs the results to the `output.txt` file.

The config file is used to configure FPGADefender and the tools it uses. The configuration file is parsed using the Python's ConfigParser package and therefore it consists of sections and options. The configuration file should have the following items specified:

- virus_signatures: Names of the virus signature packages to be executed
  - Specific virus_signature options described in the next section
- connection_attributes: Optional section for adding attributes to connections
  - attributes_file: Path to the CSV file describing which connections get which attributes.
- removables
  - connections_file - Path to a text file describing which connections should be removed from the implementation graph before the scans.

The different available virus signatures can be set up in the config file by adding the name of the virus signature class under the virus_signatures section, as shown in Table 1.

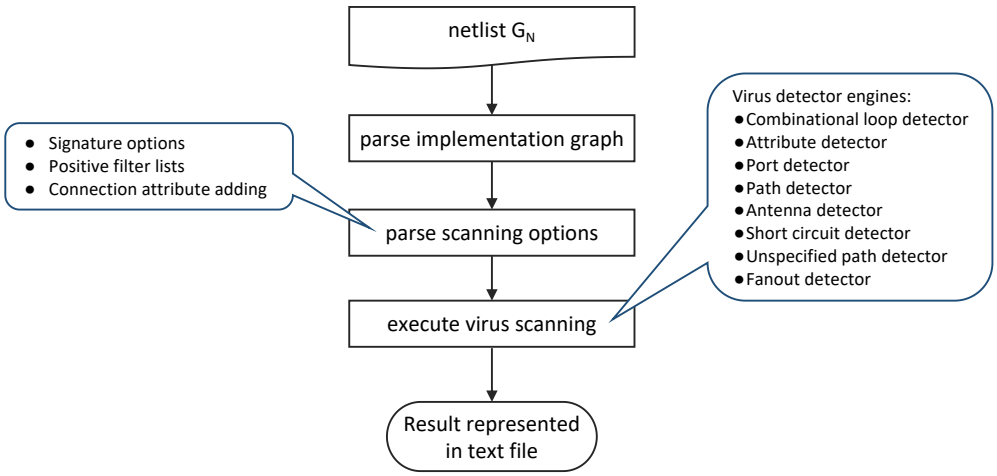

Figure 14: FPGADefender flowchart.

Table 1: Virus signatures in FPGADefender.

Feature	Signature	Options	Description
Ring oscillator detection	CombinatorialLoopDetector	ring_oscillator_detection - ignored_attributes_file	This detects loops in the given implementation
Disallowed port detection	PortDetector	node_detection - disallowed_nodes_file	Can detect disallowed port usages, like snooping on neighbouring designs
Disallowed path detection	PathDetector	path_detection - disallowed_begin_nodes_file & disallowed_destination_nodes_file	Can detect disallowed path usages like paths next to leaky long wires
Short circuit detection	ShortCircuitDetector	short_detection - short_location_file	This detects outputs with multiple used inputs which can cause short circuits
Antenna detection	AntennaDetector	antenna_detection - allowed_input_antennas_file & allowed_output_antennas_file	Can detect undesired dangling input and output wires
Unspecified path detection	UnspecifiedPathDetector	unspecified_path_detection - specified_begin_nodes_file & specified_end_nodes_file & specified_routing_nodes_file	Can detect paths which start or end at specified ports but use disallowed routing ports. The detected paths will be from the end ports which don't start at the specified start ports
Fan-out detection	FanOutDetector	fan_out_begin_nodes_file - fan_out_begin_nodes_file & fan_out_end_nodes_file	Can detect all nodes which are connected to too many end nodes. The threshold is set to 100 currently
Attribute detection	AttributeDetector	-	Can detect all nodes with the attribute "LATCH"

To build the executable, firstly a requirements file has to be set for the venv environment variable. With this, we can run:

```
pip install -r requirements.txt
```

This will install the executables using the `PyInstaller` tool got from pip. When building the executable, we have to make sure to add the virus scanner packages given in the config file as hidden imports, as shown in the following example:

```
pyinstaller virusscanner/__main__.py -n
    virusscanner -F --hidden-import=
    virusscanner.parsing.signatures.
    ring_oscillator_detection
```

To add more than one signature, the `.spec` file can be modified.

5 SCAN RESULTS

We ran FPGADefender on a benchmark of malicious bitstreams and this section presents briefly the results. As a sanity check, we

also ran scans on bitstreams that do not contain malicious circuits and FPGADefender had not reported any issue, except for one case: a true random number generator that actually uses ring-oscillators as a source of randomness. In detail we provide the following reports:

- Combinatorial loop and transparent latch detection are reported in Figure 15. The file lists a couple of cycles detected. Each cycle starts with a status line stating the specific class of ring-oscillator. FPGADefender supports detecting ROs through LUTs, cascading multiplexers (MUX7/MUX8 in Xilinx notation), CLA carry logic, DSP blocks and latches. After this the entire first cycle of each class is reported. This can be identified by the first and last entry of each cycle pointing to the same node.
- Short-circuits are reported in Figure 16. This section reports first the number of short circuit situations found and then list for the first detected switch matrix multiplexer the input ports activated. Each switch matrix multiplexer can only connect to no port (if not used) or to at most one of its available inputs.
- Latches are reported in Figure 17. This section reports latches used in cycles but also all other latches which are not malicious, but which indicates that the bitstream was not implemented following good RTL design principles.
- Antennas are reported in Figure 18. The report lists the last port of an antenna which allows investigating the antenna issue using the Vivado tool suite.
- Fan-outs are reported in Figure 19. The fan-out report lists the nets with the highest fan-out in the design. The number of nets reported is specified in the config file.

6 CONCLUSIONS AND DISCUSSION

In this tutorial we provided a small survey on recent FPGA hardware security research and we revealed that in particular ring-oscillators impose a real world threat. With this, we described how the academic tool GoAhead can be used to build a Time-to-Digital Converter for UltraScale+ FPGAs which was used for evaluating a larger number of ring-oscillator designs. This resulted in one design that has the enormous waste power potential of over 2kW on an Alveo U200 data center card and experiments on that board resulted in a power-induced crash using just 15% of the available LUT resources of the available VU9P FPGA. In the reminder of this tutorial, we showed how the open-source tool FPGADefender can detect (probably all kinds of) ring oscillator designs for mitigating this threat.

The huge waste power potentials point out that hardware Trojans and other malicious circuits are a real threat and only very little logic is required to crash a system. We like to stress that this is not a vendor-specific problem and the threats discussed in this tutorial apply to any FPGA from any vendor. However, we also showed that malicious circuits can be detected automatically and that this is even possible at the bitstream level. We believe that security through some level of virus scanning is inevitably needed as part of an FPGA ecosystem. We also believe that such security tools

```
# Output for "Mali_04_LUTs_RO_I3.json" : ROs from LUT6 primitives
CombinatorialLoopDetector: 2000.0
  Found the following cycles:
  INT_X1Y126 IMUX_W22 ->
  CLEM_X1Y126 CLE_CLE_M_SITE_0_A4 ->
  CLEM_X1Y126 CLE_CLE_M_SITE_0_A_O ->
  INT_X1Y126 LOGIC_OUTS_W0 ->
  INT_X1Y126 INT_NODE_IMUX_32_INT_OUT0 ->
  INT_X1Y126 IMUX_W22
  ...

# Output for "Mali_07_MUX7_RO.json" : ROs from MUX7 primitives
CombinatorialLoopDetector: 2000.0
Found the following cycles:
  INT_X3Y76 INT_INT_SDQ_62_INT_OUT1 ->
  INT_X3Y76 INT_NODE_GLOBAL_4_INT_OUT0 ->
  INT_X3Y76 INT_NODE_IMUX_41_INT_OUT1 ->
  INT_X3Y76 BYPASS_W4 ->
  INT_X3Y76 INODE_W_1_FT1 ->
  INT_X3Y76 BOUNCE_W_0_FT1 ->
  CLEM_X3Y76 CLE_CLE_M_SITE_0_AX ->
  CLEM_X3Y76 CLE_CLE_M_SITE_0_BMUX ->
  INT_X3Y76 LOGIC_OUTS_W17 ->
  INT_X3Y76 INT_NODE_SDQ_71_INT_OUT1 ->
  INT_X3Y76 INT_INT_SDQ_62_INT_OUT1
  ...

# Output for "Mali_09_CLA_ADD_RO.json" : ROs from Carry Logic primitives
CombinatorialLoopDetector: 2000.0
  Found the following cycles:
  INT_X11Y79 BYPASS_W4 ->
  INT_X11Y79 INT_NODE_IMUX_43_INT_OUT0 ->
  INT_X11Y79 IMUX_W18 ->
  CLEM_X11Y79 CLE_CLE_M_SITE_0_A6 ->
  CLEM_X11Y79 CLE_CLE_M_SITE_0_AMUX ->
  INT_X11Y79 LOGIC_OUTS_W21 ->
  INT_X11Y79 INT_NODE_SDQ_76_INT_OUT0 ->
  INT_X11Y79 INT_INT_SDQ_62_INT_OUT1 ->
  INT_X11Y79 INT_NODE_GLOBAL_4_INT_OUT0 ->
  INT_X11Y79 INT_NODE_IMUX_41_INT_OUT1 ->
  INT_X11Y79 BYPASS_W4
  ...

# Output for "Mali_10_DSP_RO.json" : ROs from DSP primitives
CombinatorialLoopDetector: 2880.0
  Found the following cycles:
  INT_X3Y36 INT_NODE_IMUX_13_INT_OUT0 ->
  INT_X3Y36 IMUX_E34 ->
  DSP_X3Y35 DSP_DSP48_1_C12 ->
  DSP_X3Y35 DSP_DSP48_1_FAKE_PREG ->
  DSP_X3Y35 DSP_DSP48_1_XOROUT2 ->
  INT_INTF_R_X3Y38 LOGIC_OUTS_R11 ->
  INT_INTF_R_X3Y38 LOGIC_OUTS_L11 ->
  INT_X3Y38 LOGIC_OUTS_E11 ->
  INT_X3Y38 INT_NODE_SDQ_15_INT_OUT1 ->
  INT_X3Y38 SS2_E_BEG3 ->
  INT_X3Y36 SS2_E_END3 ->
  INT_X3Y36 INT_NODE_IMUX_13_INT_OUT0
  ...

# Output for "Mali_11_Latch_RO.json" : ROs from transparent latches
CombinatorialLoopDetector: 2000.0
  Found the following cycles:
  INT_X16Y99 IMUX_W21 ->
  CLEM_X16Y99 CLE_CLE_M_SITE_0_D6 ->
  CLEM_X16Y99 CLE_CLE_M_SITE_0_DQ ->
  INT_X16Y99 LOGIC_OUTS_W10 ->
  INT_X16Y99 INT_NODE_IMUX_38_INT_OUT1 ->
  INT_X16Y99 IMUX_W21
  ...
```

Figure 15: Report sample for combinatorial loop detection.

```
# Output for "elegant_short.SHORT.json" : Short-circuit design
ShortCircuitDetector: 1.0
INT_X3Y1 EE12_BEG0 has the following inputs which can cause a short:
  INT_X3Y1 NN12_END0
  INT_X3Y1 EE12_BLN_7_FT1
  INT_X3Y1 NN4_E_BLN_7_FT1
  INT_X3Y1 EE4_W_END0
  INT_X3Y1 EE4_E_END0
  INT_X3Y1 NN12_END1
  INT_X3Y1 NN4_W_BLN_7_FT1
  INT_X3Y1 EE12_END0
```

Figure 16: Report sample for short-circuit detection.

```
# Output for "Mali_11_Latch_RO.json" : Latches attribute detection
CombinatorialLoopDetector: 2000.0
Found the following cycles:
  INT_X16Y99 IMUX_W21 ->
  CLEM_X16Y99 CLE_CLE_M_SITE_0_D6 ->
  CLEM_X16Y99 CLE_CLE_M_SITE_0_DQ ->
  INT_X16Y99 LOGIC_OUTS_W10 ->
  INT_X16Y99 INT_NODE_IMUX_38_INT_OUT1 ->
  INT_X16Y99 IMUX_W21
  ...
AttributeDetector: 2000.0
  CLEM_X16Y99 CLE_CLE_M_SITE_0_DQ
  ...
  2000 latches are used!
```

Figure 17: Report sample for transparent latch detection.

```
# Output for "antennas.json" : Antenna report
AntennaDetector:
Found the following dangling output ports:
  INT_X1Y0 INT_NODE_IMUX_15_INT_OUT1
```

Figure 18: Report sample for antenna detection.

```
# Output for "elegant_short.SHORT.json" : Fan-out report
FanOutDetector:
  CLEL_R_X2Y51 CLE_CLE_L_SITE_0_E_O has a fan-out of: 97
  CLEM_X4Y38 CLE_CLE_M_SITE_0_AQ2 has a fan-out of: 96
  CLEM_X4Y38 CLE_CLE_M_SITE_0_BQ has a fan-out of: 95
  CLEM_X4Y38 CLE_CLE_M_SITE_0_CQ has a fan-out of: 94
  CLEL_R_X14Y10 CLE_CLE_L_SITE_0_H_O has a fan-out of: 91
  CLEM_X13Y14 CLE_CLE_M_SITE_0_AQ has a fan-out of: 90
  CLEM_X9Y1 CLE_CLE_M_SITE_0_C_O has a fan-out of: 9
  CLEM_X6Y68 CLE_CLE_M_SITE_0_F_O has a fan-out of: 9
  ...
```

Figure 19: Report sample for fan-out detection.

can reliably solve any security issue and that even multi-tenancy in datacenters is well possible. For industry, the best way to address security challenges is by opening architectures, bitstreams, and tools in order to give the research community best possibilities to develop mitigation strategies.

With this tutorial, we want to create awareness for FPGA security and stimulate research to ensure that FPGA security will be treated in a proactive manner.

7 ACKNOWLEDGMENTS

This work is kindly supported by the National Cyber Security Centre of the UK through the project *rFAS - reconfigurable FPGA Accelerator Sandboxing* (grant agreement 4212204/RFA 15971) and by the European Commission through the H2020 project *EuroEXA* (grants 754337).

We also thank the Xilinx University Program for tools and boards.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

[6] 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)*.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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

[14] 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.

[15] 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.

[16] 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)*.

[17] 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.

[18] 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)*.

[19] 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)*. 13–24.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] S. Trimberger and J. Moore. 2014-08. FPGA Security: Motivations, Features, and Applications. *Proc. IEEE* 102, 8 (2014-08), 1248,1265.

[25] 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.

[26] A. Wild, A. Moradi, and T. Güneysu. 2018. GliFreD: Glitch-Free Duplication Towards Power-Equalized Circuits on FPGAs. *IEEE Trans. Comput.* 67, 3 (2018), 375–387.

[27] 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.

[28] 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.

[29] 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.