LoopBreaker: Disabling Interconnects to Mitigate Voltage-Based Attacks in Multi-Tenant FPGAs

Hassan Nassar*, Hanna AlZughbi†, Dennis R. E. Gnad*, Lars Bauer*, Mehdi B. Tahoori*, Jörg Henkel*
**Karlsruhe Institute of Technology (KIT), Institute for Computer Engineering (ITEC)*
{hassan.nassar,dennis.gnad,lars.bauer,mehdi.tahoori,henkel}@kit.edu
†*Independent*, alzughbi.hanna@gmail.com

Abstract—**FPGAs are being offered in the cloud as accelerator resources that can be shared among multiple users (i.e. tenants). Recently, various approaches have shown that fault attacks launched from one tenant region to another are possible, leading to timing faults or crashes of the FPGA. It is, therefore, important that malicious tenants are limited in their ability to cause such security problems. So far, the existing countermeasures against such attacks check the configuration bitstreams before they are reconfigured. Such offline approaches have various practical limitations, e.g. they may force the tenants to unveil their design secrets.**

In this paper, we present LoopBreaker, a novel runtime solution that can disable the entire activity of a malicious tenant region, in order to rapidly stop a potential attack before it results in a crash (i.e. Denial-of-Service). We implemented and tested multiple attack types and found that realistic attacks demand at least 12–$26\,\mu s$ to be successful. A partial reconfiguration to overwrite the malicious tenant region demands $200\,\mu s$ in our real-world implementation, which is too slow to prevent the attack from leading to a crash. Instead, our proposed LoopBreaker method only needs $1.5\,\mu s$ to stop a malicious tenant, which makes it the first online approach that can successfully stop challenging voltage drop-based attacks from causing a crash.

I. Introduction

Since a few years, FPGAs are made available by cloud providers as reconfigurable accelerators [1], [2], further pushing the limits of computing efficiency through heterogeneous computing. Additionally, multi-tenant access to FPGAs can be used for higher utilization per device, leading to higher efficiency in the overall operation of the datacenter [3]–[5]. That is done by partitioning the reconfigurable fabric of the FPGA into multiple regions, namely, a *static region* and *Partially Reconfigurable Regions (PRRs)*. While the static region contains supervisory logic to conduct management tasks (e.g. reconfiguring the PRRs), the PRRs contain the tenant-specific designs. A PRR can be reconfigured at runtime (by uploading a partial bitstream), without changing the static region or disrupting other tenants. That feature effectively allows the cloud provider to virtualize the FPGA resources.

However, FPGA-based multi-tenant systems may be vulnerable to security threats, which need to be addressed before widespread adoption of these systems becomes feasible in the cloud [6], [7]. While security threats on the logical level can be formally verified and checked in the design [8], [9], threats on the physical level are much harder to assess. FPGA-internal fault attacks can even provoke a Denial-of-Service (DoS), i.e. 'crash', of the FPGA, affecting the availability of the FPGA to all tenants. These attacks cause massive voltage fluctuations inside the FPGA by maliciously utilizing FPGA resources targeting weaknesses in the respective power supply [10]–[14]. Existing countermeasures check the design of a tenant before allowing to upload it on the FPGA, which either can be extremely restrictive or allows certain attacks to escape from being detected [15], [16]. Additionally, they may impose practical limitations regarding tenants Intellectual Property (IP) disclosure to cloud providers.

In this paper, **we present LoopBreaker, a novel solution that can preemptively disable all activities in a malicious PRR**, in order to rapidly stop a potential attack before it results in a crash, i.e. DoS. Our solution is implemented in the static part of a multi-tenant FPGA design, where voltage sensors are used to detect malicious activities to trigger our countermeasure.

We investigate a novel and very fast method to disable all interconnects in a PRR, which makes LoopBreaker radically faster than reconfiguring the PRR with an empty design (which is not fast enough to stop such attacks). Recent countermeasures aim at disabling all clocks of the malicious PRR to stop its activity [17]. However, that does not help against malicious PRRs that use self-oscillating circuits instead of relying on external clocks [16], which is also undetected by countermeasures that check the design before reconfiguring it. To the best of our knowledge, our proposed LoopBreaker is the only feasible solution that can successfully stop such attacks.

Our main contributions can be summarized as follows:

- The first online (i.e. runtime) countermeasure against voltage-based fault attacks, in multi-tenant FPGAs, which can also handle self-oscillating attacks (that do not rely on a clock).
- A novel reconfiguration-based approach to disable all interconnects in the malicious PRR, radically faster than reconfiguring it with an empty design, without disturbing the other legitimate tenants.
- The first exhaustive evaluation of various voltage drop-based fault injection attempts on multi-tenant FPGAs.

II. Background and Related Work

A. Dynamic Partial Reconfiguration

Dynamic Partial Reconfiguration (DPR) is used in many areas like reconfigurable processors [18], [19] and application specific accelerators [20]–[22]. It allows the designer to have a static part and a dynamic part that can be reconfigured as required. The designer allocates PRRs in the dynamic part and can have multiple configurations (e.g. multiple tenant-specific designs) for the same PRR that need to have the same interface to the static part. For each configuration of the PRR, a *partial bitstream* and an additional *blank configuration* are generated [23]. The blank configuration contains an empty design that can be used when the PRR is not needed, to reduce its power. The DPR can be done externally, through JTAG, or internally, using the Internal Configuration Access Port (ICAP) and a configuration manager (such as [24], [25]).

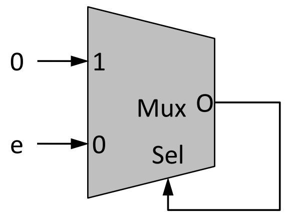
For Multi-tenant FPGAs, DPR is a crucial part [5], [26]. The system manager generates a design that contains a static part and different blank PRRs. Then, as tenancy requests come in, the PRRs are populated by the different tenant custom designs.

B. Fault Attacks on Cloud FPGAs

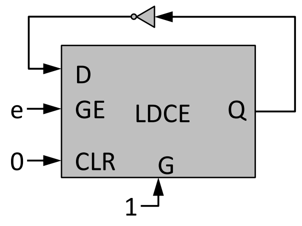
Traditionally, fault attacks on Integrated Circuits (ICs) are carried out by direct physical access to the respective system. By causing shorter clock periods or voltage drops in the Power Distribution Network (PDN), timing violations and thus faults can occur in the IC. If executed with the right precision, various attacks can be performed, such as skipping security checks or extracting cryptographic secret keys using differential fault analysis [27]. With the introduction of cloud FPGAs, it became feasible for attackers to also cause faults inside an FPGA itself, without the need of physical access or external devices [10], [12].

Fault attacks from inside the FPGA can be performed by causing a strong voltage drop using power wasting circuits, like Ring Oscillators (ROs), that are implemented using standard FPGA primitives [10]. These ROs have their own *internal frequency*. On top of this RO-internal frequency, the ROs are toggled on and off with a lower frequency, typically within a range of kHz to MHz. By maliciously orchestrating that *toggling frequency*, to match the frequency response of the PDN, the power supply of an FPGA board can be attacked to eventually crash the entire platform [10]. With more fine-grained control, it has also been shown that timing faults can be injected in other circuits on the same FPGA device [12], [13], where one work in [28] also showed that, depending on the toggling frequency, either timing faults or crashes can be caused, while keeping the remaining parameters unchanged. The combinational loops in these works [10], [12], [13], [28] are easily detected by offline analysis, as some cloud platforms and publications have already demonstrated in [11], [15], [16], [29]. However, it is also feasible to cause faults using other types of power wasting circuits that are harder or even infeasible to detect [11], [14], [16].

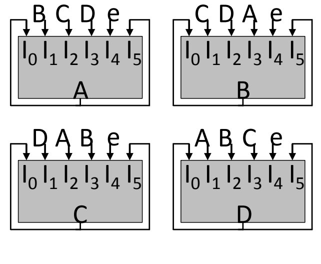
In Figure 1, we show four power-wasting circuit types that cause high voltage drops and subsequent crashes or timing faults. The first three power wasters, Figure 1(a), (b) and (c), all exhibit a self-oscillating behavior, and at least (b) is hard to detect [11], [16]. Figure 1(a) uses muxes to perform the oscillation by feeding back its output as selection. The inputs to the mux are opposite to their order, i.e. if input port 0 is selected, then the output will be 1 (assuming that the enable signal ‘e’ is 1), and vice versa. Figure 1(b) uses a latch to accomplish the oscillating behavior, by adding an inverter in the path that is fed back into the input. The constructs in Figure 1(a) and (b) do not get detected as combinational loops by Xilinx Vivado, as they use other primitives than Look-Up Tables (LUTs). Only the Enhanced Ring Oscillator (ERO) in Figure 1(c) is easily detected by Vivado and Amazon [7], [11], [16]. In that specific LUT configuration, an inverter is implemented through I_5 that can be disabled or enabled through I_4. Each four replicas of this basic block are combined together, the outputs are then propagated to the other input pins of each LUT. This does not affect the output of the LUT but rather wastes more power by distributing these outputs. Vivado detects the ERO, however, it is the strongest power waster found in the literature [16]. Hence, it is also used to evaluate LoopBreaker as it will have the highest impact among all the power wasters. Note that, for the rest of the paper, ERO will be called RO for brevity. The fourth attack in Figure 1(d) is based on dual-port RAM [14] and is not detected by [16]. It creates short circuits rather than self-oscillation, which is achieved by writing data via one port and simultaneously writing the inverse data to the same address via the other port.



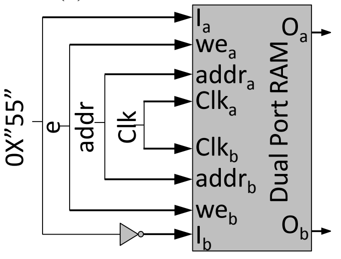
(a) Mux based attack



(b) Latch based attack



(c) ERO based attack



(d) RAM-based attack

Fig. 1: Different types of attacks implemented for the experiments, suggested in [11], [14]–[16]

C. Countermeasures to FPGA-internal Fault Attacks

Since most attacks mentioned in Section II-B are based on combinational loops, scanning FPGA bitstreams for the respective circuit variants would prevent such attacks, as different publications on *bitstream checking* have shown [15], [16]. By extracting the netlist of the bitstreams, predefined patterns of malicious structures, suspected as fault injection circuits, can be detected. However, these publications have

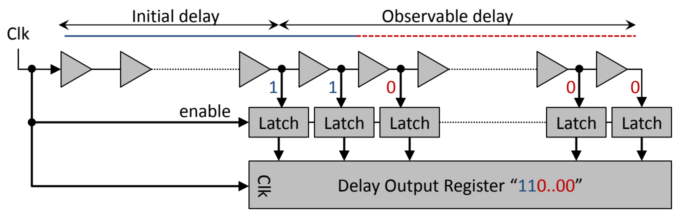


Fig. 2: Concept of TDC-based voltage sensor based on [10]

also mentioned that it might not be feasible to check all bitstreams for all variants, since they might contain possible vendor IPs or use potentially malicious circuits for benign purposes. Furthermore, new types of fault injecting circuits are still feasible, as it has been shown using the aforementioned method that uses dual-port RAMs [14] or benign modules [30].

A recently proposed active countermeasure, against voltage drop-based fault attacks, uses multiple voltage sensors to detect the location from where the malicious logic is causing the voltage drop, and then disable that part by using clock gating [17]. While their proposed location detection is an interesting approach that can also be integrated into our approach, their clock gating countermeasure can not disable all malicious tenants. For instance, malicious tenants can use self-oscillating circuits to generate their own clock, totally bypassing the countermeasure used in [17]. Instead, our proposed LoopBreaker method is also able to successfully prevent attacks that do not rely on external clock signals.

D. *Detecting Voltage drop-based Fault Attacks*

This paper is mainly focusing on stopping an attack upon detection. For basic detection, we use a high-speed voltage fluctuation sensor (shown in Figure 2), as previously introduced by Zick et al. [31] and also used in [10]. By using standard FPGA primitives, a Time-to-Digital Converter (TDC) can be implemented. That circuit can measure fine-grained transistor delays with a delay line of buffers, typically implemented with fast carry-chain primitives. To sense voltage fluctuations, a clock signal from the FPGA is fed as input to the TDC. At the same time, the same clock is fed to latches that are connected between the buffers. Since transistor delays are to some part dependent on voltage, the clock signal will propagate differently in each of its own clock cycles, being proportional to the voltage in that respective clock cycle. The sensor chain consist of two parts. The first part of the chain is used as an initial delay line, since it will always be traversed and will thus not show any voltage fluctuations. The second part of the chain is used as an observable delay line.

Wasting high power causes high voltage load on the PDN and consequently increases transistor delay, resulting in propagation delay in the input signal, which is immediately detected by the output register.

III. Methodology

LoopBreaker is our proposed method to stop the attacks presented in Section II-B. Upon activation of an attack, a voltage drop-sensor immediately detects it (as described in Section II-D) and triggers the reconfiguration of our carefully designed LoopBreaker bitstream (see Section III-C) via partial reconfiguration, to disable all interconnects of the attacker PRR. Locating the attacker PRR is out of scope of this work, but solutions like [17] can be used for this purpose. In the following, we first explain our attacker model and then show our method and how it can prevent crashes due to attacks.

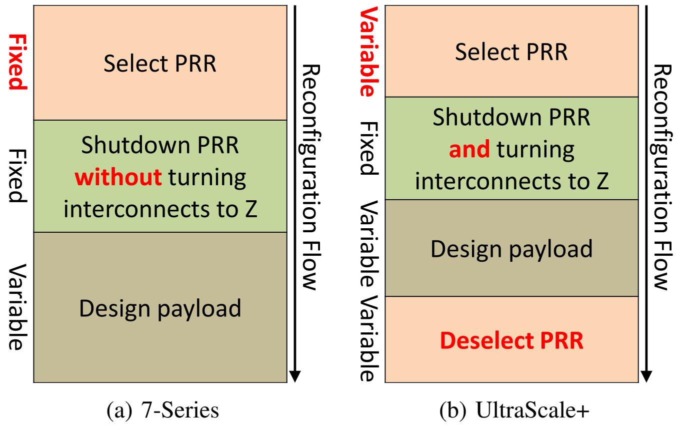


Fig. 3: Partial bitstream structure of 7-Series and UltraScale+ FPGA (major differences shown in red bold font)

A. *Attacker Model*

The attacker model in this work follows the multitude of previously published works on electrical attacks on multi-tenant FPGAs. These works aim at causing faults inside FPGA fabric [6], [7], [10]–[17], [28], which was also mentioned in the context of side-channel attacks in cloud FPGAs [6], [28].

In this work, we assume a multi-tenant FPGA setting, where both victim and attacker have their own PRR that they can use to load their own partial bitstream. The goal of the attacker is to cause a crash (i.e. DoS) or to affect the integrity of the victim's computations by causing timing faults using a malicious design, as explained in Section II-B.

B. *Reverse Engineering Xilinx' Bitstream Structure*

The most straightforward countermeasure against voltage drop-based attacks would be to use the blanking bitstream, provided by the FPGA vendor tools, that contains the 'blank configuration' mentioned in Section II-A. This countermeasure is, unfortunately, not effective in stopping such attacks, as it is too slow (i.e. slower than the attack).

To better understand this and to develop our LoopBreaker countermeasure, the structure of the bitstream has to be understood. The explanation here focuses on the UltraScale+ and the 7-series architectures from Xilinx [32], [33], however, it also holds true for other Xilinx FPGAs.

It is worth mentioning that the required understanding of the bitstream structure is not documented by the FPGA vendors. They keep these details abstracted, as the bitstream structure is not relevant for typical users. However, some previous works [34], [35] present a detailed analysis for the 7-series FPGAs. We extend that analysis to (i) include UltraScale+ FPGAs and (ii) provide a deeper understanding of the shutdown mechanism, which is a vital part of our proposed method.

Figure 3 shows the bitstream structure of (a) the 7-series and (b) the UltraScale+ FPGAs. For the 7-series, the first part of the bitstream *selects* the PRR region that shall be reconfigured by writing data to the *frames* (i.e. the smallest addressable entity of the configuration data). This selection always requires a fixed size in the bitstream, no matter how large the PRR is or how the design actually looks like. The next part of the bitstream deals with *shutting down* the selected PRR, which is done by the `SHUTDOWN` command that disconnects the interface between the static logic and the PRR. That means that the logic inside the region actually continues executing until it is overwritten by the *design payload*, i.e. the final part of the bitstream. The design payload has a variable size, as it scales with the size of the PRR. Hence, the blanking bitstream for the 7-series architecture has to overwrite a large portion of the PRR before a potential attack is stopped, which comes with long and unpredictable timing costs. As we show in our evaluation (in Section V), it is too slow (around 200 µs) to successfully prevent a potential attack.

For the UltraScale+ (Figure 3 (b)), the bitstream structure has some important differences. Firstly, the part that selects the PRR (that shall be reconfigured) no longer has a fixed size, but its size scales with the size of the PRR. Additionally, there is a new variable-sized *deselection* after the design payload that is not documented by Xilinx. The last major difference is that the shutdown process now uses the `AGHIGH` command instead of the `SHUTDOWN` command that was used in the 7-Series.

According to Xilinx user guides [32], [33], the `AGHIGH` command turns all interconnects of the selected PRR into a high impedance state 'Z', instead of only disconnecting the interface between the static logic and the PRR, as done for the 7-Series. That is actually sufficient to stop an attack in the PRR, even before the design payload of the blanking bitstream is reconfigured. However, there is still a very high likelihood that an attack is successful even before a countermeasure can execute the `AGHIGH` command. The reason is that, after reconfiguring a malicious bitstream, the attack can become active as soon as its design payload is reconfigured, i.e. even before the reconfiguration of the deselect block is completed. Even if the voltage drop-sensor immediately detects the attack and triggers the countermeasure (e.g. reconfiguring the blank bitstream), it might be executed too late. The deselect block of the malicious bitstream and the select block of the blanking bitstream need to be reconfigured before the `AGHIGH` command is executed, which requires a significant delay. These undocumented insights that we obtained via complex reverse engineering allowed us to understand why reconfiguring the blanking bitstream is too slow to prevent a crash and to realize how we could improve upon that.

A possible improvement of the blanking bitstream is the usage of bitstream compression [23], which uses the `multiframe` option of bitstreams. Normally, the data for a configuration frame is sent sequentially (in chunks of 32 bits) via the ICAP to an FPGA-internal register, before it is copied to the actual configuration bits in the FPGA. After the data of one frame is written to the register, it can be copied to multiple frames (given that they all use the same configuration data) without sending new data via the ICAP in between. This reduces the time needed for uploading the blanking bitstream, as most of its frames use the same data. However, from our tests, this did not lead to any significant improvement. Reconfiguration became five times faster, but this was still not sufficient to stop a broad range of attacks, as we show in our evaluation (in Section V). Thus, an even faster method is needed to be able to prevent crashes.

C. LoopBreaker: Disabling Interconnects

As the method that is supported by the tool vendor (i.e. the blanking bitstream, explained in Section III-B) is too slow, we aim at disabling the interconnects as fast as possible. We achieve this by generating a carefully designed LoopBreaker bitstream. Other works already studied bitstream composition [34] and even successfully generated bitstreams with a smaller size [35] by partitioning the design payload of regular bitstreams into multiple smaller bitstreams. That allowed them to fulfill latency constraints of the reconfiguration process in real-time scenarios.

Instead, our approach ignores the payload and focuses on using the `AGHIGH` command to change all interconnects into 'Z' state. To be able to do this in a flexible way, we separate each part of the bitstream (i.e. select, shutdown, payload and deselect, shown in Figure 3) into a separate bitstream. Hence, selection, deselection, shutdown and payload can be configured individually when needed (and no longer need to be configured altogether), which allows us to precisely control the configuration status of a potentially malicious PRR. However, just splitting an existing bitstream into its individual parts does not lead to valid bitstreams. Some carefully selected synchronization/desynchronization steps need to be added to each part to make it a valid bitstream. Some parts of the desynchronization have to be used in our generated bitstreams as well in order to obtain a functional design. Other parts of the desynchronization need to be omitted, otherwise the effect of turning the interconnects into 'Z' will be reversed. For example, the desynchronization contains the `GRESTORE` and the `DGHIGH` commands, both of which reverse the interconnects to normal state. Additionally, `NOP` commands need to be inserted at very specific points. After applying certain commands, a certain amount of `NOPs` have to be added. When too few are used, it leads to an error, and when too many are used, it leads to an unnecessary delay[1].

After correctly using the commands, the CRC checks of the bitstream data need to be treated properly. As the detailed CRC calculation rules are not documented, these checks need to be disabled when manipulating the bitstreams. However, simply disabling the CRC calculation does not work, because several commands require a specific CRC check. To identify these commands, a detailed analysis had to be performed. After

[1]Due to space limitations, we cannot plot the full data of the generated bitstreams in this paper. Therefore, our tool that automatically generates all needed bitstreams is available as open source, along with a technical documentation of the bitstream structure: https://git.scc.kit.edu/CES/LoopBreaker

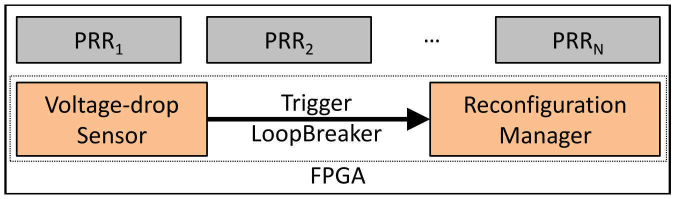


Fig. 4: Multi-tenant FPGA with sensor and reconfiguration manager included

identifying all these commands, the required CRC check can be replaced by CRC reset commands. Simply removing the CRC check commands does not work, as the bitstream would no longer function correctly.

Based on the knowledge gathered by the analysis of the bitstream structure, we were able to generate the different bitstreams needed for our solution (i.e. selection, shutdown and deselection). The most important one is the shutdown bitstream that disables the interconnects, which we call LoopBreaker bitstream. There are two slightly different versions of this bitstream for the 7-series and the UltraScale+ with 89 and 310 commands, respectively. This length difference stem from the fact that for each of them the number of `NOPs` following each command varies greatly. Another difference is, as mentioned above, that the 7-series does not perform the `AGHIGH` command, but the `SHUTDOWN` command. Hence after applying the LoopBreaker bitstream for the 7-series, no further DPR can be performed. After all tenants finish their processing, the FPGA needs to be reconfigured with the full bitstream. Nevertheless, the crash is avoided and the tenants can continue to work. For UltraScale+ FPGAs, subsequent DPRs can be performed normally.

Figure 4 shows the full multi-tenant system that we use with LoopBreaker. The connections to external components (e.g. RAM) are not shown as they are not needed for the following explanations. When a tenant is reconfigured into a PRR, we skip reconfiguring its deselect part, in order not to miss an attack in case it starts attacking immediately. In this way, the PRR is still selected, which allows us to disable it quickly if needed. The voltage drop-sensor monitors the system and, upon sensing an attack, notifies the reconfiguration manager, which then reconfigures our LoopBreaker bitstream to disable the PRR. In case that no attack was detected, during which another PRR reconfiguration shall be performed, then we first reconfigure the deselect bitstream, before reconfiguring the new bitstream. In case there are hints that a specific tenant is malicious (e.g. from offline analysis of the bitstream; see Sections II-B and II-C), then we can also reconfigure the select bitstream for that specific PRR, in order to be prepared for an attack, but that is beyond the scope of this work. Depending on the provided features of the utilized reconfiguration manager, modifications might need to be made to it and we explain our choice in Section IV.

IV. Experimental Setup

We use a Xilinx ZCU102 board that contains a Zynq UltraScale+ FPGA to implement our solution, as shown in Figure 5. We have also adopted the same solution on a Xilinx VC707 board that contains a Virtex-7 FPGA, thus demonstrating that LoopBreaker can be utilized for different FPGA types. However, given the fact that the UltraScale+ architecture is the most recent one of the two and more likely to be used in cloud environments, we mainly focus on the ZCU102 board and show the respective results in Section V.

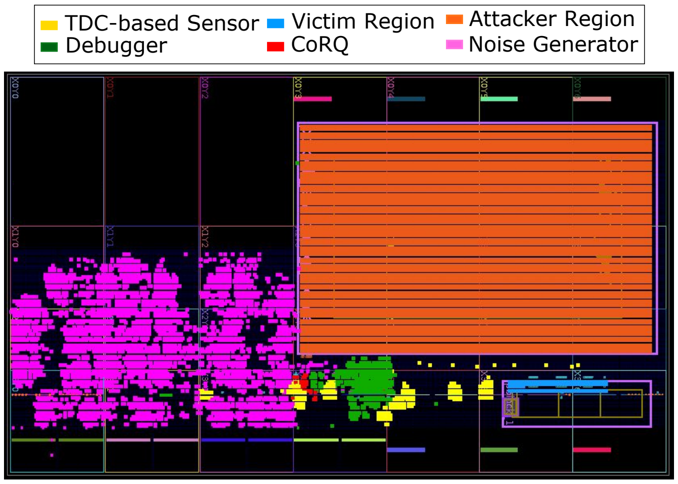


Fig. 5: Floorplan of the experimental setup on the FPGA

Our experimental setup consists of two PRRs and a static design. One PRR corresponds to the attacker and the other one is the victim. As for the victim, we use three redundant adders with almost zero remaining slack (0.011–$0.013\,\mathrm{ns}$) to represent complex designs that fully utilize their timing constraints. Such zero-slack designs are most sensitive against fluctuations (e.g. voltage drop or clock jitter) and thus, when an attack occurs, the results of the adders will be affected by timing faults. The adders get the same input (we made sure that they cannot be removed by optimizations) and thus should produce the same output. By comparing their outputs, we can detect timing faults, which allows us to evaluate whether or not an attack was stopped fast enough before timing faults occurred.

For the attacker PRR, we implemented the four state-of-the-art attacks presented in Section II-B and Figure 1 and a normal design (i.e. without attacker) that consists of 10 instances performing JPEG encoding and decoding in parallel. Moreover, the attacker PRR contains a clock generator, based on a self-oscillating structure. Hence, it cannot be stopped by the method from [17]. The attacker PRR can use up to 48% of the resources available on the FPGA. We have implemented the attacker in a way that allows us to configure how many of its available resources and which toggling frequency (see Section II-B) it should use for an attack. To evaluate different attacker scenarios, we sweep the toggling frequency from $10\,\mathrm{Hz}$ to $50\,\mathrm{MHz}$ and the *attacker size* from 0 to $48\,\%$ of the available FPGA area.

The static logic of the entire design (see Figure 4) consists of multiple parts. As voltage drop-sensor, we use the Time-to-Digital Converter (TDC) from [6], [10], as it can detect all significant voltage fluctuations (see Section II-D). To implement them in Xilinx FPGAs, we use the Carry4 primitive for 7-series and the Carry8 primitive for UltraScale+ FPGAs, as it was done in previous works [6], [10]. The sensors are calibrated to have a threshold that can be used to differentiate between normal activity of the logic on the FPGA or malicious activity. The sensor consist of a chain of buffers, where 14 bits are used as initial delay and 64 bits are used as observable delay. The clock used for the TDC sensor is 200 MHz.

As reconfiguration manager, we use the Command-based Reconfiguration Queue (CoRQ) [24] to perform the reconfigurations of the partial bitstreams. It is available as open source and it offers several features that we can use to make our solution as fast as possible. For instance, it allows to store bitstreams either on-chip or off-chip. We use the provided on-chip storage for our LoopBreaker bitstreams to achieve the highest reconfiguration bandwidth, whereas regular tenant bitstreams may be stored off-chip. Moreover, as CoRQ is capable to enqueue different reconfiguration jobs, we can use that to implement our countermeasure. For instance, when reconfiguring a PRR on an UltraScale+ FPGA, we do not reconfigure the deselect block (see Figure 3), but we only enqueue a bitstream for the deselect block, in order to reconfigure it later, before any other PRR is reconfigured. In case that an attack is detected beforehand, we can abort the queued reconfiguration and still have the most-recently configured PRR selected. For the ZCU102 and the VC707 boards, CoRQ operates at 200 MHz and 100 MHz, which are the highest frequencies that are supported by the ICAP interfaces of the UltraScale+ and the 7-series FPGAs, respectively.

The other parts of the static logic are implemented only for the experiments and tracking the results. The first part is a noise generator, based on the ISCAS'89 benchmark circuits [36], which is used to simulate the behavior (i.e. fluctuating activity that can also lead to slight voltage drops) of other PRRs or other components that may exist in the system. As final part, we use a debugger that consists of an Integrated Logic Analyzer (ILA) core and a Universal Asynchronous Receiver-Transmitter (UART) communication module. The ILA core tracks various signals from the system which helps to evaluate the results, e.g. the comparison between adders with zero slack and the behavior of the sensor signals. As for the UART module, it is used to communicate the different test scenarios to the FPGA from an external source.

V. Results

At first, we experimented with the JPEG encoder/decoder in the attacker PRR. As expected, no significant voltage drops were detected by the sensors and therefore no false positives were recorded for our experiments. When using the RAM-based attack (from Figure 1(d)), the sensor does not identify any voltage drops. However, the RAM-based attack does not lead to any timing faults or crashes when using up to 48 % of the available FPGA area (i.e. the maximum area it can use when there is a second tenant of the same size). Only with an attacker size of 98 % of the FPGA, the RAM-based attack caused a few crashes (in 5 % of the cases), but no timing faults were detected. As the RAM-based attack requires nearly the entire FPGA area to be successful, it does not leave any space for a second tenant and thus it is irrelevant for a multi-tenant system, so we excluded it from further analysis.

In the following, we focus on the other three attack types (shown in Figure 1(a)-(c)) and first present and analyze the results of attackers that precisely know which toggling frequency has the highest probability of a crash/fault, which corresponds to the worst-case scenario for any kind of countermeasure. Afterwards, we evaluate attackers without such knowledge, where they would try various toggling frequencies that lead to a lower crash/fault probability.

A. Attacker with Precise Knowledge about the Most-Destructive Toggling Frequency (Worst Case Scenario)

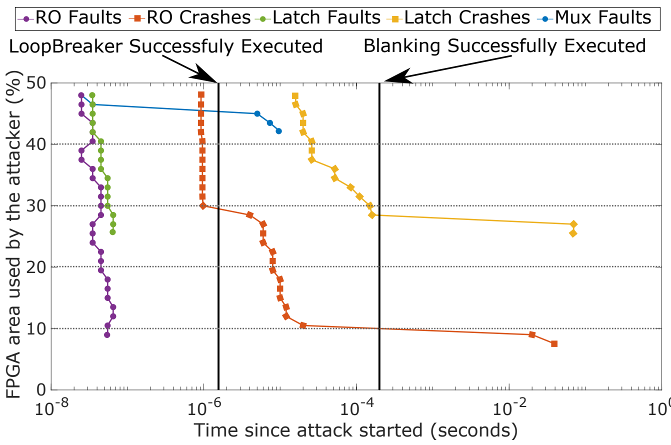


Fig. 6: Fastest observed latency until an attack leads to timing faults or a crash in comparison to the countermeasure execution time of our LoopBreaker (1.5 µs) and the Blanking Bitstream (200 µs). Countermeasure execution times longer than the attack latency will lead to faults/crashes. For scenarios where the attacker can use at most 30 % of the FPGA area, LoopBreaker can prevent all crashes.

LoopBreaker and Blanking require 1.56 µs and 200 µs, respectively, to successfully stop an attack. Note that we use bitstream compression (as explained in Section III-B) for the Blanking solution, otherwise it would take longer (i.e. 1 ms). Figure 6 shows the latencies until an attack leads to a crash or a timing fault. For each attack type and attacker size (i.e. Y-axis in Figure 6), we have tested different toggling frequencies, repeated the experiments 20 times and reported the fastest observed time until a fault/crash occurred. As a general trend, a larger attacker size typically needs a shorter time for the attack to be successful . The two vertical lines in Figure 6 (i.e. LoopBreaker and baseline Blanking) show the time needed from the start of the attack until the solution successfully stops it.

Attacks that are faster than the countermeasure, cannot be prevented. Note that we did not only *calculate* whether or not a countermeasure should theoretically prevent an attack (by comparing times), but we experimentally tested that, by running the attack and the automated detection and prevention

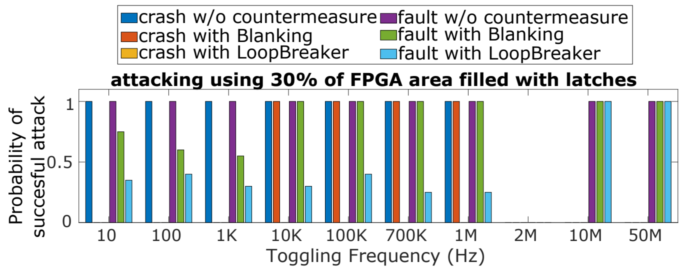


Fig. 7: Probability of successful attacks (crashes and timing faults) when using different countermeasures (bars) against an attacker that uses 30 % of the FPGA area for latch-based attacks (see Figure 1(b)) with different toggling frequencies (X-axis). Our LoopBreaker solution achieves 0 % crashes and a significantly reduced number of timing faults.

on the FPGA boards (as shown in Section IV). Figure 6 shows that the Blanking solution could only prevent a small portion of the crash attacks, whereas LoopBreaker can stop most of them. Only crashes due to RO-based attacks that use attacker size larger than 30 % of the available FPGA area, could not be prevented by LoopBreaker. To the best of our knowledge, no online method exists that can prevent these attacks. However, RO-based attacks can be easily detected by offline methods, i.e. before even reconfiguring the malicious tenant to the FPGA. The more realistic latch-based attacks would all be prevented by LoopBreaker, whereas Blanking was too slow for most of them. Note that attacker sizes larger than 50 % could lead to even faster attacks, however, that would not leave enough space for a same-sized second tenant and thus is irrelevant for multi-tenant systems.

Most of the timing faults occur so fast following the start of the attack. Not even LoopBreaker could prevent them. However, no timing fault went undetected by the used sensor. Therefore, the malicious tenant could be stopped as fast as possible to prevent any additional faults in the other tenants' region. Additionally, the detection of the attack (and thus the increased likelihood of timing faults) is reported to the system manager, which can then inform the tenants to take appropriate measures (e.g. rollback in case they were not protected by redundancy measures like Triple-Modular Redundancy (TMR), but this is beyond the scope of this work).

B. Attacker without Knowledge about the Most-Destructive Toggling Frequency (Average Case)

So far only the worst case attack scenario was considered. However, attackers might not be in possession of an FPGA with the same setup as the one in the cloud environment. Therefore, they cannot perform a full characterization and thus do not know the most destructive toggling frequency. Figure 7 shows the effect of different toggling frequencies on the probability of an attack leading to a crash or timing fault. The attacks considered here use enhanced latches (from Figure 1(b)) and an attacker size of 30 % of the available FPGA area. Altogether, we evaluated 30 different toggling frequencies. Due to space limitations, we highlighted the results of 10 different frequencies in Figure 7. We selected these 10 frequencies to represent the decades from 10 Hz up to 100 MHz. Additionally, we added measurement results for 700 kHz and 2 MHz, due to their distinctive behavior that is worth mentioning.

At 700 kHz, the attack remarkably causes faults but no crashes, however, sometimes a crash happens but only after the attack has been stopped. At 2 MHz, surprisingly, no crashes or timing faults where observed at all. The reasons for this are not clear, but generally such high frequencies in the MHz range seem to be less effective to create crashes, e.g. at 10 MHz and 50 MHz no crashes where observed at all, even without using a countermeasure. It is noticeable that LoopBreaker does not suffer from any crashes in the benchmarked scenarios (latch-based attacks with attacker size of 30%) at all. Additionally, LoopBreaker could significantly reduce the probability of timing faults, compared to the Blanking solution, for the wide range from 10 Hz to 1 MHz.

Table I shows the detailed results for different combinations of attack type, countermeasure type, attacker size and toggling frequencies. In addition to the latch-based attacks, at attacker sizes of 30 % (as shown in Figure 7), we add the results of RO-based attacks and different attacker sizes, ranging from 7.5 % to 45 % (i.e. the biggest size that leaves enough space for a second same-sized tenant). Due to the limited success of Mux-based attacks (i.e. leading to no crashes and much less timing faults than RO-based or Latch-based attacks), they are excluded from this analysis for brevity. Each probability is calculated based on 20 runs of the specific combination. Table I shows two cases for each scenario: An *average case* where the attacker uses a random toggling frequency, and the *worst case* where the most-destructive toggling frequency is used.

By looking at the evaluation of crashes (in the upper half of Table I), it is noticeable that our LoopBreaker countermeasure is at least as good as the Blanking countermeasure, and most of the time is even better. For RO-based attacks, LoopBreaker can prevent all crashes up to an attacker size of 22.5 %, whereas Blanking is only able to prevent crashes up to an attacker size of 7.5 % (i.e. an attacker that uses 3 times less area). For latch-based attacks, LoopBreaker can even prevent crashes in all evaluated scenarios, whereas Blanking can only prevent crashes up to an attacker size of 25.5 %.

From the evaluation of faults (in the lower half of Table I), it is apparent that for RO-based attacks, neither Blanking nor LoopBreaker performs better than no countermeasure. As discussed in Section V-A and Figure 6, most of the timing faults occur so fast after the start of the attack that not even LoopBreaker could prevent them, but the attack is detected and reported to the system manager to take appropriate measures (beyond the scope of this work). For Latch-based attacks with random toggling frequency (average case), LoopBreaker can reduce the probability that an attack leads to timing faults compared to Blanking. However, the most important achievement is the significant reduction of crashes compared to Blanking, allows multi-tenant systems to remain operational, even when an attacker uses more than 22.5 % of the available

TABLE I: Probability of successful crash and fault, depending on the attack type, countermeasure type and attacker size

	Attack Type	**Countermeasure**	**Attacker size for Ring Oscillator (RO)-based attacks**					**Attacker size for Latch-based attacks**		
			7.5%	15%	22.5%	30%	45%	25.5%	30%	45%
Probability that the Attack leads to a Crash	Attack with most-destructive toggling frequency (worst case)	No countermeasure	40%	100%	100%	100%	100%	100%	100%	100%
		Blanking	0%	100%	100%	100%	100%	0%	100%	100%
		LoopBreaker	0%	**0%**	**0%**	100%	100%	0%	**0%**	**0%**
	Attack with random toggling frequency (average case)	No countermeasure	9.3%	70%	70%	100%	100%	19.5%	70%	70%
		Blanking	0%	40%	40%	70%	100%	0%	40%	40%
		LoopBreaker	0%	**0%**	**0%**	**40%**	100%	0%	**0%**	**0%**
Probability that the Attack leads to a Timing Fault	Attack with most-destructive toggling frequency (worst case)	No countermeasure	0%	100%	100%	100%	100%	100%	100%	100%
		Blanking	0%	100%	100%	100%	100%	100%	100%	100%
		LoopBreaker	0%	100%	100%	100%	100%	100%	100%	100%
	Attack with random toggling frequency (average case)	No countermeasure	0%	90%	90%	100%	100%	39.5%	90%	90%
		Blanking	0%	90%	90%	100%	100%	39.5%	79%	90%
		LoopBreaker	0%	90%	90%	100%	100%	**20%**	**42.5%**	90%

FPGA resources and precisely knows the most-destructive toggling frequency.

VI. Discussion

Even though our countermeasure appears to be simple, now that we investigated and explained how it can be done, it has to be highlighted that: (i) it was challenging and non-straightforward to reverse-engineer and test all the information required to actually develop a fully functional countermeasure, (ii) it is the by far fastest existing countermeasure against voltage drop-based attacks that can even handle attacks that use self-oscillating circuits, and (iii) it is very valuable for the community to have full access to all our presented insights and to our fully functional solution that we will make available as open-source release.

LoopBreaker does not have the ability to completely prevent all timing faults, however, it can noticeably reduce their probability. A small amount of timing faults can still occur before LoopBreaker removes the malicious bitstream. We think that with the current FPGA technology, it is not possible to completely prevent all timing faults in case of an attack. That makes it even more valuable that LoopBreaker can prevent all crashes from the more realistic Latch-based attacks. In addition, the existence of a voltage drop with the increased likelihood of a timing fault is detected by the voltage sensor and can be used by the system to take corresponding measures (e.g. rolling back to a checkpoint in case the design was not protected by TMR etc., but this is beyond the scope of this work).

As mentioned in Section IV, LoopBreaker operates at the maximum frequency that ICAP supports (i.e. 200 MHz). Moreover, it performs only 310 commands, which are the minimum number of commands needed to disable the interconnects. Hence, to the best of our knowledge, there is no room for improving the performance of LoopBreaker, other than future support from FPGA vendors, for a faster way to disable interconnects.

The results show that our LoopBreaker countermeasure can stop a large number of attacks or at least reduce their success probability. While some types of RO-based attacks can already be prevented through bitstream checking, by cloud providers [16], [29], our LoopBreaker can prevent all crashes from the more realistic Latch-based attacks, and can even prevent some faults introduced with Muxes.

VII. Conclusion

Multi-tenant FPGAs are of increasing interest in improving the computing efficiency in the cloud. However, it has been shown recently that they can be affected by voltage-based attacks. Hence, mechanisms to counteract such attacks are needed before widespread adoption of such services. Existing mechanisms focus either on bitstream checking before being loaded into the FPGA, or detecting malicious activity at runtime. In this paper, we introduced our LoopBreaker solution that uses partial reconfiguration to quickly disable malicious tenants, upon detection, using a voltage fluctuation sensor. Our proposed method only needs 1.5 µs to stop a malicious tenant, which makes it the first online approach that can successfully stop challenging voltage drop-based attacks from causing a crash.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

We also acknowledge Moritz Hauer who ported the TDC-based voltage sensors from Xilinx 7-series to Ultrascale(+).

References

[1] Amazon Web Services, "EC2: Elastic Compute Cloud." [Online]. Available: http://aws.amazon.com/ec2/

[2] "Alibaba Cloud – Cloud Computing Services," 2021. [Online]. Available: https://www.alibabacloud.com/de/product/computing

[3] S. A. Fahmy, K. Vipin, and S. Shreejith, "Virtualized FPGA accelerators for efficient cloud computing," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 430–435.

[4] A. Khawaja, J. Landgraf, R. Prakash *et al.*, "Sharing, protection, and compatibility for reconfigurable fabric with AmorphOS," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 107–127.

[5] J. M. Mbongue and C. Bobda, "Accommodating multi-tenant FPGAs in the cloud," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 214–214.

[6] O. Glamočanin, L. Coulon, F. Regazzoni, and M. Stojilović, "Are cloud FPGAs really vulnerable to power analysis attacks?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2020, pp. 1007–1010.

[7] T. La, K. Pham, J. Powell, and D. Koch, "Denial-of-service on fpga-based cloud infrastructures — attack and defense," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, pp. 441–464, Jul. 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8982

[8] B. Ustaoğlu, K. Schmitz, D. Große, and R. Drechsler, "Recofused partial reconfiguration for secure moving-target countermeasures on FPGAs," *SN Applied Sciences*, vol. 2, no. 8, pp. 1–17, 2020.

[9] S. Trimberger, "Trusted design in FPGAs," in *Design Automation Conference (DAC)*, 2007, pp. 5–8.

[10] D. R. E. Gnad, F. Oboril, and M. B. Tahoori, "Voltage drop-based fault attacks on FPGAs using valid bitstreams," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–7.

[11] T. Sugawara, K. Sakiyama, S. Nashimoto *et al.*, "Oscillator without a combinatorial loop and its threat to FPGA in data centre," *Electronics Letters*, vol. 55, no. 11, pp. 640–642, 2019.

[12] J. Krautter, D. R. E. Gnad, and M. B. Tahoori, "FPGAhammer: Remote voltage fault attacks on shared FPGAs, suitable for DFA on AES," *Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, no. 3, 2018.

[13] D. Mahmoud and M. Stojilović, "Timing violation induced faults in multi-tenant FPGAs," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2019, pp. 1745–1750.

[14] M. M. Alam, S. Tajik, F. Ganji *et al.*, "RAM-Jam: Remote temperature and voltage fault attack on FPGAs using memory collisions," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2019, pp. 48–55.

[15] J. Krautter, D. R. Gnad, and M. B. Tahoori, "Mitigating electrical-level attacks towards secure multi-tenant FPGAs in the cloud," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 12, no. 3, pp. 1–26, 2019.

[16] T. M. La, K. Matas, N. Grunchevski *et al.*, "FPGADefender: Malicious self-oscillator scanning for Xilinx UltraScale+ FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 3, pp. 1–31, 2020.

[17] G. Provelengios, D. Holcomb, and R. Tessier, "Mitigating voltage attacks in multi-tenant FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2021.

[18] M. Damschen, M. Rapp, L. Bauer, and J. Henkel, *i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems*. Springer, 01 2020, pp. 1–36.

[19] S. Vassiliadis, S. Wong, G. Gaydadjiev *et al.*, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363–1375, 2004.

[20] A. Hassan, H. Mostafa, H. A. H. Fahmy, and Y. Ismail, "Exploiting the dynamic partial reconfiguration on NoC-based FPGA," in *New Generation of CAS (NGCAS)*, 2017, pp. 277–280.

[21] R. Chaves, G. Kuzmanov, and L. Sousa, "On-the-fly attestation of reconfigurable hardware," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2008, pp. 71–76.

[22] K. Salah, "An area efficient multi-mode memory controller based on dynamic partial reconfiguration," in *IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2017, pp. 328–331.

[23] *Partial Reconfiguration User Guide*, Xilinx, 2015.

[24] M. Damschen, L. Bauer, and J. Henkel, "CoRQ: enabling runtime reconfiguration under WCET guarantees for real-time systems," *IEEE Embedded Systems Letters*, vol. 9, no. 3, pp. 77–80, 2017.

[25] L. Pezzarossa, M. Schoeberl, and J. Sparsø, "A controller for dynamic partial reconfiguration in FPGA-based real-time systems," in *International Symposium on Real-Time Distributed Computing (ISORC)*, 2017, pp. 92–100.

[26] J. Krautter, D. R. Gnad, F. Schellenberg *et al.*, "Active fences against voltage-based side channels in multi-tenant FPGAs," in *International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[27] B. Yuce, P. Schaumont, and M. Witteman, "Fault attacks on secure embedded software: Threats, design, and evaluation," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 111–130, 2018.

[28] D. R. E. Gnad, J. Krautter, M. B. Tahoori *et al.*, "Remote electrical-level security threats to multi-tenant FPGAs," *IEEE Design & Test*, vol. 37, no. 2, pp. 111–119, 2020.

[29] "AWS Developer Forums: [DRC LUTLP-1] Combinatorial Loop Alert," 2018. [Online]. Available: https://forums.aws.amazon.com/thread.jspa?messageID=851556

[30] G. Provelengios, D. Holcomb, and R. Tessier, "Power wasting circuits for cloud FPGA attacks," in *International Conference on Field-Programmable Logic and Applications (FPL)*, 2020, pp. 231–235.

[31] K. M. Zick, M. Srivastav, W. Zhang, and M. French, "Sensing nanosecond-scale voltage attacks and natural transients in FPGAs," in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2013, pp. 101–104.

[32] *7 Series FPGAs Configuration User Guide*, Xilinx, 2015.

[33] *UltraScale Architecture Configuration User Guide*, Xilinx, 2020.

[34] SymbiFlow Team. Project X-Ray. [Online]. Available: https://prjxray.readthedocs.io/en/latest

[35] E. Rossi, M. Damschen, L. Bauer *et al.*, "Preemption of the partial reconfiguration process to enable real-time computing with FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 2, Jul. 2018.

[36] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *International Symposium on Circuits and Systems,*, 1989, pp. 1929–1934 vol.3.