IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 2, pp. 426–450. DOI:10.46586/tches.v2024.i2.426-450

JustSTART: How to Find an RSA Authentication Bypass on Xilinx UltraScale(+) with Fuzzing

Maik Ender[1], Felix Hahn[1], Marc Fyrbiak[1], Amir Moradi[2] and Christof Paar[1]

[1] Max Planck Institute for Security and Privacy, Bochum, Germany
firstname.lastname@mpi-sp.org
[2] Technische Universität Darmstadt, Darmstadt, Germany
amir.moradi@tu-darmstadt.de

Abstract. Fuzzing is a well-established technique in the software domain to uncover bugs and vulnerabilities. Yet, applications of fuzzing for security vulnerabilities in hardware systems are scarce, as principal reasons are requirements for design information access, i.e., HDL source code. Moreover, observation of internal hardware state during runtime is typically an ineffective information source, as its documentation is often not publicly available. In addition, such observation during runtime is also inefficient due to bandwidth-limited analysis interfaces, i.e., JTAG, and minimal introspection of hardware-internal modules.

In this work, we investigate fuzzing for Xilinx 7-Series and UltraScale(+) FPGA configuration engines, the control plane governing the (secure) bitstream configuration within the FPGA. Our goal is to examine the effectiveness of fuzzing to analyze and document the opaque inner workings of FPGA configuration engines, with a primary emphasis on identifying security vulnerabilities. Using only the publicly available hardware chip and dispersed documentation, we first design and implement `ConFuzz`, an advanced FPGA configuration engine fuzzing and rapid prototyping framework. Based on our detailed understanding of the bitstream file format, we then systematically define 3 novel key fuzzing strategies for Xilinx FPGA configuration engines. Moreover, our strategies are executed through mutational structure-aware fuzzers and incorporate various novel custom-tailored, FPGA-specific optimizations to reduce search space. Our evaluation reveals previously undocumented behavior within the configuration engine, including critical findings such as system crashes leading to unresponsive states of the whole FPGA. In addition, our investigations not only lead to the rediscovery of the recent starbleed attack but also uncover a novel unpatchable vulnerability, denoted as JustSTART (CVE-2023-20570), capable of circumventing RSA authentication for Xilinx UltraScale(+). Note that we also discuss effective countermeasures by secure FPGA settings to prevent aforementioned attacks.

Keywords: FPGA, FPGA Configuration Engine, FPGA Security, FPGA Bitstream Protection, Hardware Fuzzing, Fuzzing Framework, Vulnerability Discovery, starbleed

1 Introduction

Field Programmable Gate Arrays (FPGAs) are a foundation in modern digital system landscape as their *field-programmable* nature offers flexibility and adaptability. To realize its flexibility, the FPGA consists of 2 parts: i) the configuration engine that handles loading of a so-called *bitstream* (representing a digital gate-level design) into ii) the FPGA grid – also called fabric – that consists of millions of reconfigurable Look-Up-Tables (LUTs) and routing

Received: 2023-10-15 Accepted: 2023-12-15 Published: 2024-03-12

to run the digital design, among other reconfigurable hardware primitives. Since FPGAs are commonly deployed in security-critical systems, including industrial control systems, cloud computing systems, and even military applications, manufacturers have implemented bitstream protection systems to ensure confidentiality, integrity, and authenticity. While several works highlighted security vulnerabilities of the cryptographic implementation, i.e., via side-channel attacks [MBKP11, SMOP15, MS16], optical contactless probing [TLSB17], or implementation attacks [SW12, EMP20, ELMP22, FS19], a system-level methodical analysis of the configuration engine itself is – to the best of our knowledge – missing in the open literature. However, since the configuration engine is complex (e.g., handling multiple security modes and various commands during the initialization process) and opaque (e.g., little to no information about its implementation details is publicly known, i.e., a gray-box setting), understanding its detailed architecture and security mechanisms plays an integral role for FPGA security. Even more importantly, any vulnerability in the configuration engine is unlikely to be patched as it is implemented in hardware and thus poses a massive threat.

In recent years, fuzzing in the software domain has seen widespread adoption in both academia and industry due to its effectiveness in uncovering software vulnerabilities. In particular, *feedback-driven fuzzing*, i.e., generation of a random input mutation and observing system behavior that is then feed back into the input generation, has been shown to be effective. Even though various fuzzing methods have been proposed to analyze hardware systems [TSC+21], they leverage design information such as Hardware Description Language (HDL) source code to perform its analysis and thus do not work for settings where only the manufactured chip is available with limited observable system information.

Goals and Contributions In this paper, we focus on fuzzing for Xilinx 7-Series and UltraScale(+) FPGA configuration engines. Inspired by the aforementioned starbleed vulnerability in the configuration engine and capabilities of software fuzzing methods, our goal is to answer the following research question:

> *To what extent can we leverage systematic fuzzing techniques to derive (security) implementation information from the opaque configuration engine of Xilinx 7-Series and UltraScale(+) FPGAs?*

In order to answer this research question, we first want to highlight that we – as a security research community – only have a superficial understanding of implementation details based on incomplete and dispersed publicly available limited documentation (gray-box setting). Note that this generally challenges effectiveness of fuzzing as interpretation of observed behavior is limited. In addition, effectiveness of fuzzing is another key challenge as we are generally limited by the device connection, e.g., via USB/JTAG. From a high-level point of view, our goal is to increase the public knowledge about configuration engine implementation details with a focus on *how* certain security functionalities are implemented. Therefore, our approach and contribution are as follows

- **`ConFuzz` FPGA Configuration Engine Fuzzing Framework (Section 3)** We design and implement `ConFuzz`, a mutational bitstream fuzzing framework on the basis of a rapid prototyping approach to define bitstreams in a declarative manner. With context-specific FPGA optimization, such as structure-aware bitstream mutations using a bitstream grammar and auto-generated encryption blocks, we improve the efficacy of our approach (that is generally limited by JTAG speed). We publicly released `ConFuzz` under the MIT license on GitHub [Emb23].
- **Fuzzing Strategies (Section 4)** We develop three main strategies to systematically evaluate the 7-Series and UltraScale(+) series configuration engine: i) bitstream

structure, ii) inter command, and iii) intra command. While the first strategy analyzes the general bitstream structure, the latter two concentrate on the commands.

- **JustSTART (Section 5.2)** Our evaluation uncovers a new unpatchable vulnerability named *JustSTART* (CVE-2023-20570) that bypasses the RSA authentication of Xilinx UltraScale(+) FPGAs and thus enables attackers to load trojanized or modified bitstreams. We want to note that this attack can be mitigated when both FPGA bitstream encryption and authentication are enabled.
- **Further Security Vulnerabilities & Understanding (Section 5)** In further case studies on the 7-Series and UltraScale(+), we automatically uncover the recent starbleed attack and various undocumented behavior, for example, crashing the FPGA into an unresponsive state or an undocumented RSA test mode. We want to highlight that based on our findings, we also contribute to the understanding of the opaque configuration engine.

2 Background & Related Work

To understand the mechanics of the Xilinx UltraScale(+) FPGAs configuration process, we now provide some fundamental FPGA background aspects. Afterward, we detail related work focusing on hardware and embedded device fuzzing.

2.1 Xilinx UltraScale(+) FPGAs

FPGAs are a class of Integrated Circuits (ICs) containing re-programmable logic to enable users to change a digital design even after manufacturing. To this end, FPGAs consists of the so-called *fabric*, i.e., grid of reconfigurable hardware elements such as LUTs, Digital Signal Processors (DSPs), Block RAMs (BRAMs), I/O, and reconfigurable routing connecting all elements. From a high-level perspective, the bitstream consists of a header to handle the configuration and the fabric configuration data containing the proprietary encoding of the gate-level digital design description. Since we are only interested in the configuration process, we refer the interested reader to diverse publications and open-source frameworks [Sym17, Not08, ESW+19, DWZZ13, NR08] that deal with the fabric configuration data. We want to note that the header (starting with a sync word `0xAA995566`) consists of commands and data to read and write to the configuration engine registers.

Configuration Process According to the user guide *UG570* [Xil23], Xilinx UltraScale(+) FPGAs are configured by loading the application-specific configuration data into the internal memory of the fabric. This configuration is done by loading the bitstream through the configuration engine via 1 of 5 possible configuration interfaces into the device, i.e., SelectMAP, Internal Configuration Access Port (ICAP), Serial Peripheral Interface (SPI), Byte Peripheral Interface (BPI), or Joint Test Action Group (JTAG). Hence, this configuration engine manages the configuration process, which is crucial for FPGA security. As noted before, bitstreams can be encrypted, so they may require decryption first. Even though the configuration engine details are not publicly known in detail, we have set up a mental model of our current understanding in Figure 1.

Configuration Packets Commands and data are organized as 32-bit words, where a packet header indicates a read, write, or NOP to a desired register and the number of words written to that register. The data written to that register follows after the header. Table 1 shows the type 1 packet header format. 11 header bits are marked as *reserved*, meaning they have no function and are reserved for future use. Type 2 headers extend

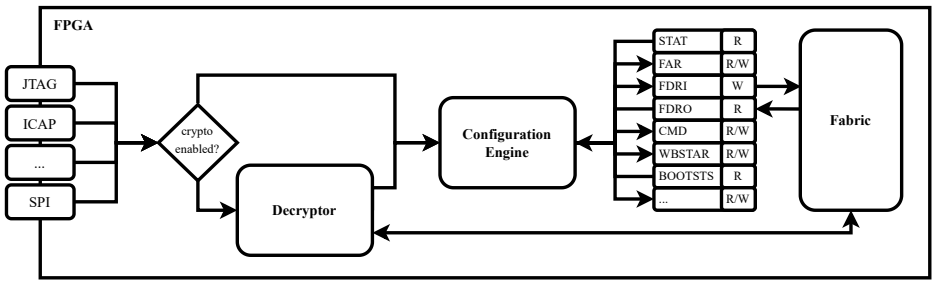

Figure 1: Our mental model of the configuration engine.

type 1 headers and lack the register address to maximize the bits used for the word count field.

Table 1: Type 1 packet header: bits marked with *R* have no functionality and are *reserved* for future use (based on [Xil23]).

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Configuration Registers The configuration engine uses the configuration registers to manage its internal state and configuration. The only way to communicate with the configuration engine is to read and write to these registers. For example, the `FDRI` register is used to write configuration data to the fabric of the FPGA, and the `STAT` register contains information about the current status of the configuration engine. 5 bits address the registers as shown in Table 1. Hence, 32 registers are addressable at most. According to the publicly available documentation, there are only 20 registers, leaving 12 undocumented registers. In anticipation of our framework `ConFuzz` in Section 3, we analyze these undocumented registers since there are indications that a subset of registers have actual use. Note that we refer to them as *unknown registers* throughout the work at hand. While most registers consume 32-bit data, they are of individual length, e.g., the `GCM-IV` register consumes 4 words (= 128-bit) of a Galois/Counter Mode (GCM) Initialization Vector (IV). Many registers allow read access to aid debugging. Moreover, readback of the fabric data is also possible if allowed by the security configuration.

FPGA Bitstream Security As noted before, FPGA bitstream protection schemes enable the security goals *confidentiality* and *authenticity*. To ensure hardware design confidentiality, the bitstream fabric data can be encrypted and readback disallowed. The bitstream authenticity ensures that the bitstream is not manipulated and no malicious design is executed on the FPGA. Note that this hinders the integration of hardware Trojans and other bitstream-level attacks.

For the UltraScale(+) FPGAs, Xilinx implemented an Rivest–Shamir–Adleman (RSA) authentication and Advanced Encryption Standard (AES) encryption mechanisms, which can be used solely or combined to ensure the aforementioned security goals. The AES used in either the GCM or Counter (CTR) mode ensures bitstream confidentiality. If the AES is used solely, it is used in the GCM mode to implement bitstream authenticity. Also, a proprietary Galois field-based checksum *X-GHASH* is implemented to ensure the authenticity of blocks of 8 words within the bitstream [ELMP22]. Besides, an RSA authentication mechanism can be utilized to ensure authenticity. It can be used with a

plaintext bitstream or an AES-encrypted bitstream. For the latter, the AES is operated in the CTR mode without the GCM and *X-GHASH* authentication. The AES keys can be stored in a battery-backed RAM (BBRAM) or burned to fuses. Similarly, an Secure Hash Algorithm 3 (SHA-3) hash of the RSA public key is stored in fuses. Two other fuses enforce that only encrypted and/or RSA-authenticated bitstreams can be loaded. In summary, three different combinations of security measures can be used to ensure the authentication and integrity of the FPGA bitstream: AES-GCM, AES-RSA, or plain-RSA.

2.2 Related Work

We now provide a brief overview of existing literature encompassing FPGA security.

FPGA Attacks Since FPGAs are a foundation for many systems, they are a commonly targeted device. Attacks against the configuration engine can be divided into 3 main areas: i) side-channel attacks [HLF+20, MBKP11, SMOP15, MS16] leverage information leaked through power consumption or electromagnetic radiation, ii) probing attacks [TLSB17, LTK+18] recover internal states or the key, and lastly iii) implementation attacks [EMP20, ELMP22, SW12, FS19] exploit vulnerabilities in the configuration engine implementation.

Hardware Fuzzing Hardware fuzzing is a subgroup of the fuzzing landscape. Analogous to software fuzzing approaches, hardware fuzzing [TSC+21, CRD+23, LGBD19, MRBC21, DCSSK21, PA22, HYJ+17, XQZ+21, EMS+22, QSD+21, EFI21, MSK+18] leverages the HDL source code to observe the hardware behavior via simulation. For example, in recent work, Canakci *et al.* [CRD+23] analyze an embedded processor implementation by monitoring the transitions in its control and status registers during simulation. Similar to our work, they rely on registers of the target device to provide feedback and guide the fuzzing process. However, in our approach, we do not have the HDL source code of the configuration engine and cannot leverage any of the aforementioned hardware fuzzing approaches. Methods for black-box or gray-box fuzzing on the hardware are scarce in the open literature. In 2010, Koscher *et al.* [KCR+10] analyzed embedded devices leveraging fuzzing, focusing the CAN bus in automotive vehicles. Since then, several works targeted embedded devices fuzzing with limited observable behavior [LCC+15, BHT14, KL13, AVR14, CFH+22]. Additionally, fuzzers targeting the Instruction Set Architecture (ISA) of processors have been proposed, for example [Dom17] uncovers hidden instructions, glitches, and vulnerabilities in x86 ISA, also aiming to understand the hardware better.

While such approaches have certain similar characteristics to our approach, e.g., no source code available and limited system behavior observation, key differences are the context in which fuzzing is carried out as the fuzzing methods are context-dependent and have to be adjusted for each scenario. Note that no prior work is dealing with fuzzing the FPGA configuration engine.

In the recent work by Easdon et al. [ESSG22], a rapid prototyping framework is leveraged to develop attacks and identify vulnerabilities. Similarly, we have adopted such rapid prototyping approach to extend our investigation and contribute to the general understanding of the configuration engine's inner workings and find vulnerabilities.

3 `ConFuzz` Framework

We now introduce the design and implementation of our FPGA configuration engine fuzzing framework `ConFuzz`. First, we describe several key challenges for FPGA configuration engine fuzzing, then we present the system architecture, workflow and fundamental components of `ConFuzz` (Section 3.1), and finally provide implementation details (Section 3.2).

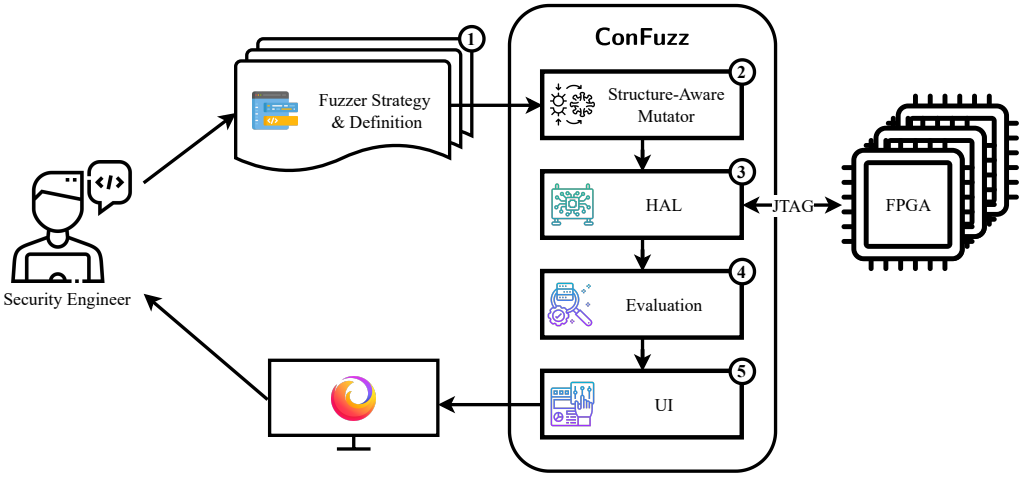

Figure 2: `ConFuzz`: Architecture and Workflow.

Key Challenges Since fuzzing of FPGA configuration engines has the speed limitation of its underlying hardware, a key challenge is test cycle performance, i.e., bitstream generation and communication with the FPGA for programming and subsequent information retrieval. A critical aspect of this challenge lies in generation of *useful*, i.e., syntactically and semantically valid, test bitstreams to facilitate effective information retrieval within our gray-box setting. Note that without such structure-aware test generation, numerous test bitstreams may be generated that do not yield any useful (security) information as these are likely invalid (e.g., missing checksum values, invalid header commands, ...).

3.1 System Architecture

`ConFuzz` is designed with high modularity and extensibility in mind. To this end, we structured `ConFuzz` upon several fundamental components, each characterized by a logically coherent feature set. In the following, we outline the workflow, providing more detail on the components, see Figure 2.

Workflow The user selects a fuzzing strategy and a target FPGA to implement a concrete fuzzer ❶. The implemented fuzzer uses a structure-aware mutator ❷ to generate modified test bitstreams automatically. Then, each modified bitstream is handled by the Hardware Abstraction Layer (HAL) ❸ in order to program the FPGA and obtain information of the hardware state after configuration. Finally, the state is post-processed in an evaluator ❹, i.e., analyzed for crashes or deviated behavior, and optionally displayed in a user interface ❺ for manual inspection.

To guide the reader, we now detail the components of `ConFuzz` in a bottom-up fashion, starting from the hardware to the user interface, as the higher-level components require context of the lower-level components. Moreover, note that we designed `ConFuzz` with the aforementioned challenges in mind.

Hardware Abstraction Layer (HAL) ❸ In order to instrument FPGA device(s), we designed a hardware abstraction layer. Key parts are communication using JTAG or Universal Asynchronous Receiver / Transmitter (UART) and managing the FPGA configuration engine state. In particular, state management is vital to infer information for fuzzing feedback (e.g., hardware-internal register values, or whether the FPGA responds at

all, ...). Note that we are in a gray-box setting and cannot inspect all hardware internals during configuration, so our goal is to infer as much direct and indirect information as possible: the Xilinx FPGA configuration engine only communicates via its configuration registers, see Section 2.1, thus we opted to read-back of all configuration registers.

Structure-Aware Mutator ❷ As identified before, a key challenge is the generation of *useful* bitstreams, i.e., the generation of syntactic and semantic valid bitstreams. We have implemented a grammar-based fuzzing approach within the structure-aware mutator, which relies on a custom-tailored FPGA bitstream format grammar. This grammar is based on the bitstream file format, incorporating all bitstream functions, ranging from simple headers and data value fields to automated generated checksums. Key components are automated encryption and authentication primitives to ensure correct handling of a bitstream block to encrypt and/or authenticate them. Within these bitstream primitives, *fuzzing masks* can be defined to automatically generate variants of the defined bitstream. For example, one can define a fuzzing mask to test 3 bits written to a register, then 2^3 bitstream would be generated testing all possible bit combinations for that field, while checksums, encryption, etc. are handled automatically.

State Evaluation ❹ Fuzzing intends to find unexpected *crashed* states. Therefore, upon the automated bitstream generation, transmission to the FPGA, and subsequent retrieval of its state after programming, the state is evaluate with careful analysis for unexpected states. Inspired by software fuzzing practices, we integrated the concept of a *crash* into our evaluation utilizing a range of crash settings. These settings encompass various criteria such as presence or absence of expected values within designated registers, and deviations from default states. For example, one approach is to probe the initial *clean* state immediately following device restart, then after transmitting a bitstream, any deviations to the state are flagged as a crashes. Note that the definition of a crash is user-defined, contingent upon the specific intentions during evaluation.

Fuzzer Strategy & Definition ❶ Utilizing the previously mentioned bitstream grammar and crash settings, the security engineer implements a fuzzer strategy. Note that in our comprehensive case studies, we implemented 71 fuzzers, each adhering to the three principal strategies outlined in Chapter 4 and 5.

UI/Viewer for Inspection ❺ In the final phase, the security engineer inspects generated crashes to derive insights from the fuzzing process, aiming to pinpoint potential bugs and vulnerabilities while enhancing understanding of the configuration engine. Results from the fuzzing process are accessible through a web interface, facilitating efficient manual analysis. This interface displays the current fuzzing process and shows information about every logged test case. It provides details such as the transmitted bitstream, current register values, and the meaning of individual register bits if publicly documented. Additionally, the interface highlights the executed crash evaluation, i.e., it shows any test case and corresponding registers and their values involved in a crash.

3.2 Implementation

We implemented key components such as the FPGA bitstream grammar and fuzzing strategies of `ConFuzz` using Python, in particular, to enable rapid prototyping during experiments and manual inspection. For FPGA bitstream response state management, we used the boofuzz SQLite database. As a fuzzing harness and viewer, we leveraged boofuzz [Per] and for communication with the FPGA, we used OpenOCD [Rat].

On Rapid Prototyping Natural language in the FPGA user guides and documentation provided by the vendors has the implicit drawback of ambiguity and misinterpretation by the reader. Thus, a key aspect during our design and implementation of `ConFuzz` was a horizontal rapid prototyping approach in order to rapidly verify or falsify certain assumptions about our mental model of the FPGA configuration engine implementation details based on the documentation. In particular, such rapid prototyping capability is advantageous for sense-making of certain behaviors, i.e., device crashes. Note that an observed state after configuration does not *explain* its implementation details. Typically, this is a manual sense-making task to link the documentation and observed outcome to a mental model of the presumed implementation, as, for example, [WAMP13] states:

> *The configuration data 314 is also encrypted if the DEC bit in the plaintext header is set. In an example implementation, if a* `TEST_MODE` *bit is asserted in the DLC, the configuration control circuit will expect 24 frames of configuration data. If the* `TEST_MODE` *bit is not asserted, then the configuration control circuit will write configuration data to the configuration memory until there is an indication that the end of the device has been reached.* [WAMP13]

Based on this public Xilinx patent description, we then leveraged the rapid prototyping capability of `ConFuzz` to explore the `TEST_MODE` bit. As the high-bits of the Decryption Length Counter (DLC) are typically 0 for standard lengths, we assume that the higher bits are used for the `TEST_MODE`. We then create a bitstream with the highest bit set to '1' and send 24 frames of configuration data. The bitstream was successfully programmed and thus revealed the presence of the `TEST_MODE` bit in commercial-of-the-shelf Xilinx UltraScale(+) FPGAs. Note that we leverage the `TEST_MODE` for optimization in our second case study (Section 5.2) as programming of whole fabric configuration data requires typically $\sim$ 20s for UltraScale(+) FPGAs while bitstreams with just 24 frames of configuration data are programmed in $\sim$ 2s, thus achieving a performance speed-up of $\sim 10\times$.

4 On Bitstream Fuzzing Strategies

It is crucial to acknowledge that our endeavor has its limitations. The speed constraints imposed by the use of hardware, i.e., the JTAG interface, prevent us from exhaustively fuzzing every possible configuration parameter. So we have to take a strategic *guided* approach to reduce the fuzzing test cases and test only *promising* ones. The available documentation at hand gives us a superficial understanding of the configuration engine such that we can build a first mental model of the configuration engine and bitstream grammar (cf. Section 2.1). Based on this information, we define three main strategies: i) bitstream structure, ii) inter command, and iii) intra command. While the first strategy challenges the general bitstream structure, the latter two concentrate on the commands themselves. Here, the inter-command strategy tests explicitly the behavior of each single command, while the intra-command strategy examines the interaction between them.

Note that the purpose of this work is to better understand the undocumented part of the configuration engine and thus find potential vulnerabilities. The first two strategies are primarily designed to uncover undocumented parts within the bitstream and enhance our mental model of the configuration engine, while the third strategy is also intended for detecting security vulnerabilities by especially targeting the interaction with the bitstream protection schemes. In the following, we discuss these strategies in detail, followed by the general implementation procedure of a fuzzer.

Fuzzer Strategy 1: Bitstream Structure Xilinx documents the bitstream structure, as discussed in the background in Section 2.1. The configuration engine is generally organized

in registers while the bitstream reads and writes to them. The bitstream is structured in header commands followed by the data and commands to be written in the specified registers (cf. Table 1).

Within this first fuzzer strategy, we target this bitstream structure in order to uncover undocumented bits and unused bit combinations. For example, the first three header bits determine the header type, e.g., 8 possible types could exist, while only two types are documented. Similarly, one unused opcode (11) exists, and eleven reserved bits.

Fuzzer Strategy 2: Intra Command The next layer of abstraction is configuration engine registers. The FPGA configuration engine has (most likely [1]) a total of 32 registers; however, only 20 of these registers have been documented in public resources, while several bits are still marked as reserved even when bitstreams generated by Vivado contain them.

Within the second fuzzer strategy (intra-command), we specifically target one of each register at a time to uncover undocumented registers, document the bit usage, and test their influence on the FPGA state. For example, we created a default intra-register fuzzer for the several completely undocumented registers. This fuzzer writes a single 32-bit data word to the fuzzed register, as the typical registers are of 32-bit size. We provide a detailed discussion of a fuzzer that implements the intra-command strategy in our latest case study presented in Section 5.3.

Fuzzer Strategy 3: Inter Command Moving beyond the analysis of the bitstream structure and individual commands, we explore the interactions between multiple commands and their potential ramifications within our last fuzzing strategy. One particular aspect of our inter-command fuzzing strategy involves investigating the bitstream protection features. For example, commands are placed around the new RSA authentication mode of UltraScale(+) devices. We provide a detailed discussion of a fuzzer that implements the inter-command strategy in our second case study presented in Section 5.2.

General Fuzzer Implementation Procedure A hallmark of our approach is the systematic procedure we can follow for the implementation of each fuzzer, regardless of the chosen strategy. The general workflow comprises the following steps:

1. **Selection of Target/Strategy:** First, target FPGA and board are selected, as well as one of the main fuzzing strategies. Based on public documentation, we can make assumptions regarding the configuration engine to pick a specific target within the configuration engine aligned with the chosen strategy. This enables systematically covering all areas of the configuration engine within a specific strategy.
2. **Fuzzer Construction:** With the selected fuzzing strategy and target in mind, a specific fuzzer is implemented.
3. **Fuzzer Execution & Crash Logging:** The fuzzer is then executed, automatically generating bitstreams, sending them to an FPGA, and probing the FPGA state. Only *abnormal* states are logged as a crash depending on user-defined crash settings.
4. **Interpretation and Refinement:** Identified crashes are then analyzed to discern their underlying causes and extend our mental model of the bitstream configuration engine. This process of interpretation informs the iterative refinement of the fuzzer and its parameters. In certain scenarios, we even employ secondary fuzzers designed to explore specific bit positions, configuration parameters, and crash settings or utilize the rapid prototyping feature of `ConFuzz`.

[1] Table 1 shows that 9 of 14 address bits are marked as reserved. With the help of a strategy 1 fuzzer, we could not find any register influencing the FPGA state, which is addressed via these 9 reserved address register bits. This concludes that only the 5 lower bits are used to address registers, hence a maximum of 32 addressable registers.

5 Case Studies

With the fuzzing strategies from Section 4 in mind, we implemented 71 fuzzers on Xilinx 7-Series and UltraScale(+) FPGAs. As we cannot discuss every finding in detail in the scope of this work, we provide an excerpt of the most interesting and impactful findings in Table 2; a complete list of all findings is available at [Emb23]. Three findings are discussed in this chapter, showcasing the implementation and investigation procedure.

With the 71 implemented fuzzers, we executed about 83 million test cases, which took approximately 2 weeks. For UltraScale(+) fuzzers, we used a single board and a small cluster of 15 FPGAs for the Xilinx 7-Series. In our experiments, we tested up to 2^{21} on UltraScale(+) and on our 7-Series cluster 2^{25} bitstreams per day, depending on the fuzzer configuration. We found 1677 crashes during evaluation. Overall, our approach facilitated an improved understanding of the bitstream file format and implementation of the configuration engine, including errors and crashes, see Table 2.

Table 2: Excerpt of our fuzzers and findings (7S: 7-Series, US+: UltraScale(+)).

Name	FPGA Family	Type	Findings
header types	US+	structure	Header type 010 always and header type 001 with opcode 11 does lead to a crash of the device (all registers are zero)
register 23	7S, US+	intra	Bit 23 (& 25) crashes the device (power cycle needed)
register 29	7S, US+	intra	Value matches FUSE_CNTL register (documented JTAG only)
starbleed	7S, US+	inter	Re-discovers Starbleed attack [ELMP22]
JustSTART	US+	inter	Discovers novel JustSTART attack

Development Boards For our experiments, we examined the following Xilinx FPGAs development boards: Basys 3 (Artix-7 XC7A35T), KCU116 (UltraScale+ Kintex XCKU5P), and OpalKelly XEM8320 (UltraScale+ Artix XCAU25P), while we also support the KCU105 (UltraScale Kintex XCKU040). All boards offer JTAG-over-USB for FPGA programming. Further, the jumpers are set to *JTAG* boot mode to prevent the FPGAs from being configured from any other bitstream source until a JTAG programming occurs.

5.1 Fuzzing Encrypted Bitstreams: How to Find Starbleed

The recent starbleed attack [EMP20, ELMP22] is a time-of-check-to-time-of-use vulnerability where bitstream commands can be executed before verifying their authenticity. Ender *et al.* discovered the attack "by a detailed study of the Xilinx official documents together with experiments" which is a manual laborious task. We employed an automated fuzzer, as outlined in this case study, to automate the process, leading to the rediscovery of the attack. Moreover, we found 3 additional registers vulnerable to the attack. Note that this case study focuses on UltraScale(+) devices. The fuzzer for the 7-Series works analogously, and results are available in our GitHub repository [Emb23]. We further assume that no RSA authentication is enforced by the security fuses, as it would prevent the attack.

The Starbleed Attack The root cause of starbleed is that manipulations of the ciphertext are detected after decryption and execution of commands. A manipulation can be inserted

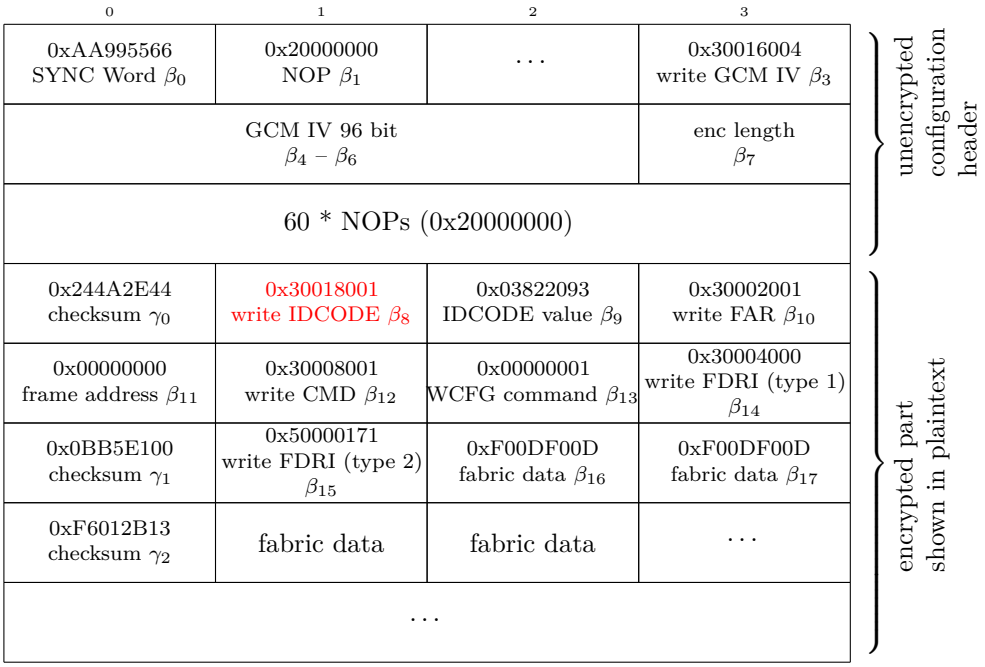

Figure 3: The encrypted bitstream utilized to be fuzzed by the starbleed fuzzer.

by flipping bits in an encrypted ciphertext undermining the used GCM mode and the underlying CTR mode malleability. An attacker can manipulate the bitstream to divert secret fabric content to the `WBSTAR` register instead. Then, when the FPGA finally detects the manipulation, the FPGA resets but does not clear said register. So it can be read out afterward. This behavior can be exploited to read out all confidential data within the bitstream via the `WBSTAR` register. For example, Figure 3 shows parts of an encrypted bitstream. β_8 is the first encrypted bitstream command. Manipulating this word would divert the following data words.

The Starbleed Fuzzers Idea In the following, we assume a scenario where the analyst has no prior knowledge about the starbleed attack but has a basic understanding of the bitstream structure from the available documentation. To detect undefined behavior, the analyst starts fuzzing an encrypted bitstream like an unencrypted one by starting with a strategy 1 fuzzer. This strategy tests the bitstream structure, e.g., the header words of the bitstream are fuzzed. We designed the fuzzer to cover all 21 active bits in the header (see Table 1), i.e., we omitted the reserved ones. We exclude the reserved bits in this particular fuzzer as the mutation space is too large, and we have already tested these bits with an unencrypted structural fuzzer, which revealed that they likely have no functionality.

In order to mimic an attacker capability who can only flip encrypted bits instead of generating correctly encrypted bitstreams, we directly mutated an encrypted bitstream rather than mutate and then encrypt the bitstream. Note that this manipulates only one word at a time, e.g., β_8 in Figure 3; no other words are changed, especially checksums and authentication tags. We created a short encrypted bitstream writing 3 frames to the fabric to reduce the fuzzed bitstream size and improve the fuzzer performance. The implementation of the starbleed fuzzer is displayed in Listing 1.

Fuzzer Execution & Analysis On a single OpalKelly XEM8320 Ultrascale+ board, the starbleed fuzzer stops execution after recording 128 crashes by default, having executed $20,119,674$ test cases in roughly 8 days. All crashes occur while mutating $\beta_1 5$ and directly reveal the starbleed vulnerability, as one can see decrypted fabric data in configuration registers. While we tried to keep the fuzzer configuration as simple as possible, resulting in an extended runtime of the fuzzer, it is plausible that a knowledgeable attacker could refine the fuzzer strategy to find the vulnerability more quickly, e.g., by modifying the starting position of the first fuzzed word or relaxing the crash settings to match other decrypted data.

On the 7-series cluster, using 15 boards in parallel, some boards fuzzed a test case range where no crashes occurred, while other boards exited early after recording 128 crashes. This results in a runtime of about 46 hours with $43,189,976$ test cases and 1156 crashes. Most crashes are similar to each other, as, for example, the same manipulations are carried out at different words in the bitstream. Again, most of them directly revealed the starbleed vulnerability, while some are not linked to starbleed, as data words are manipulated and not header words. Note that the results of the UltraScale+ starbleed fuzzer and the starbleed fuzzer on the 7-series cluster are not directly comparable due to subtle implementation differences.

With these results, the analyst would have discovered the starbleed vulnerability, i.e., manipulating an encrypted (non-RSA-authenticated) bitstream is possible as the bitstream commands are executed before the next cryptographic checksum is validated. However, the X-GHASH checksum is still an issue within the practicability of that attack because this checksum (γ_i) hinders leakage of bitstream content as it is evaluated every 8^{th} word. Nevertheless, the underlying attack foundation is found at this stage.

New Discoveries Through our fuzzing attempts, we have discovered that in addition to the `WBSTAR` register, the `TIMER`, `UNKNOWN_20`, and `BSPI` registers are also susceptible to the starbleed attack. To the best of our knowledge, this was not publicly known yet.

Accordingly, on the Virtex-6, the starbleed attack might become usable again as the `WBSTAR` register lacks the two most significant bits, i.e., the `WBSTAR` register is only 30 bits long. Unfortunately, we could not test this yet due to the lack of a legacy Virtex-6 board.

5.2 JustSTART: RSA Authentication Bypass

As of now, *no* vulnerability to the RSA bitstream authentication of Xilinx UltraScale(+) FPGAs was publicly known. Unfortunately, we discovered a flaw in said RSA authentication mechanism of Xilinx UltraScale(+) FPGAs. With the JustSTART vulnerability, the RSA bitstream is loaded to the FPGA as usual, but instead of running the RSA authentication mechanism, commands to *just start* the FPGA are inserted into the bitstream, and the device boots successfully. We discovered this flaw using our fuzzing framework `ConFuzz`. With this vulnerability, the RSA authentication of all Xilinx UltraScale(+) FPGAs can be bypassed, even if RSA is enforced by the security fuses. Currently, the only countermeasure to this attack is to enforce the bitstream encryption by the *AES only* fuse since the device does not decrypt the attacking bitstream. In the following, we describe the attack in detail and its consequences, followed by a description of finding it with `ConFuzz`.

5.2.1 RSA Authentication for UltraScale(+)

Figure 4 describes the structure of an RSA-authenticated bitstream [Xil23,Pet21,WAMP13]. The bitstream starts with an unauthenticated plaintext header followed by a write to the `RSA_DATA_IN` register. The data written to this register includes the RSA public key, SHA-3 padding, and RSA signature, calculated over the following written data, the encryption IV, decryption length counter, header commands, fabric data, and footer commands. Note here

that the RSA bitstream structure is very rigid, as the length of each written block is fixed. Even the word count for the write to the `RSA_DATA_IN` is fixed depending on the FPGA fabric size. The following bitstream data, which is not authenticated and not written to the `RSA_DATA_IN` register, is the `RDW_GO` and other commands at the bitstream footer. Note that we discovered a test mode to break out of this rigid structure (see Section 3.2).

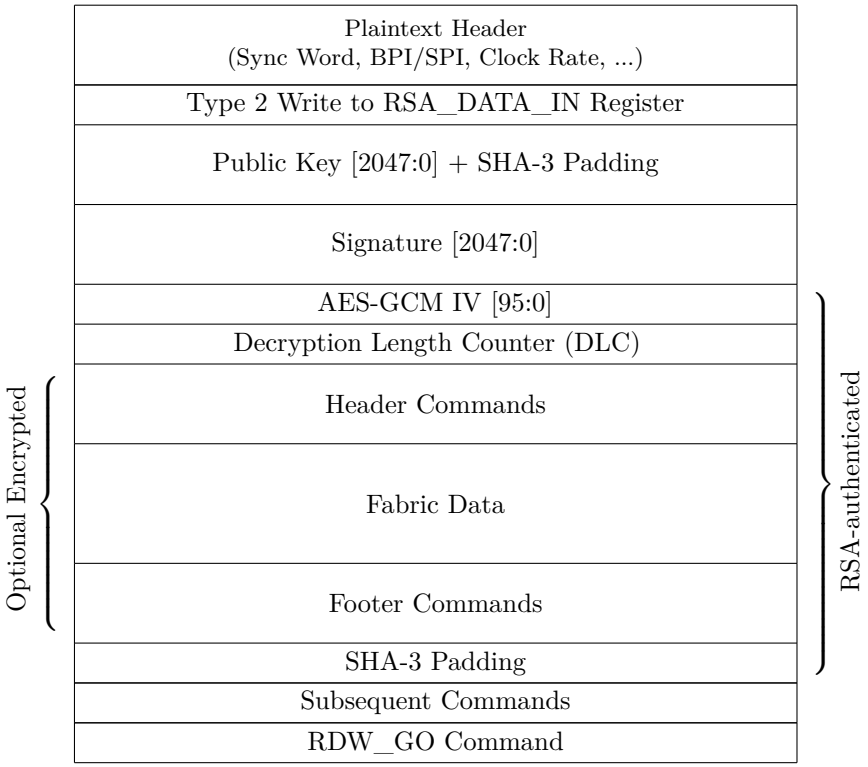

Figure 4: RSA Bitstream Structure based on [Pet21]. Gray-shaded words are written to the `RSA_DATA_IN` Register.

When using an RSA-authenticated bitstream, its authenticity is verified before activating the decryption core and decrypting it. This prevents time-of-check-to-time-of-use attacks and side-channel attacks to recover the decryption key. For this reason, the authenticated bitstream data must be buffered on the device. While the header and footer commands have a designated buffer, the fabric data is buffered in the FPGA fabric, as it would consume too much space otherwise. Hence, the write to the `RSA_DATA_IN` are writes to the header/footer buffers, as well as, the FPGA fabric. No commands are executed while writing to these buffers. The first executed bitstream commands after buffering are the subsequent unauthenticated footer commands. Then, the `RDW_GO` (Read-decrypt-write_GO) command initiates the authentication and decryption of the previously buffered data. At first, the authenticity of the bitstream is checked. If the authenticity is confirmed and the bitstream is not encrypted, the RSA header and footer are executed, starting the FPGA design. If the bitstream is encrypted, the RSA header is decrypted and executed. Afterward, the fabric is decrypted frame by frame, i.e., read from the fabric, decrypted, and written back to its place in the fabric. Finally, the RSA footer is decrypted and executed. This footer contains the commands to boot the device and run the design previously written to the fabric. In order to complete these complex tasks of authenticating and decrypting the buffered bitstream, the `RDW_GO` command needs many ticks on the configuration clock [Xil23].

5.2.2 Attack Idea

The idea of JustSTART is simple: replace the `RDW_GO` command at the end of the bitstream with the usual start-up command sequence. As a result, the authentication process is not started, and the design already present in the fabric is executed.

The attack works because the fabric is used as a buffer for the design, and the end of the bitstream, including `RDW_GO` command, is not authenticated. In the case of plaintext RSA bitstreams, the design is already present in the fabric, waiting to be authenticated by the `RDW_GO` command. If an encrypted bitstream was loaded, replacing the `RDW_GO` command prevents the activation of the decryption engine, leaving fabric data encrypted. Therefore, the attack only works if the attacker bitstream is unencrypted.

5.2.3 Attacker Model

The presented attack allows an attacker to circumvent the RSA authentication of Xilinx UltraScale(+) FPGAs. By bypassing the authentication step and *just starting* the device after the bitstream has been written to the FPGA fabric, an attacker can configure the device with arbitrary plaintext RSA bitstreams. Hence, our attacker model is the following:

- **Bitstream Manipulation** The attacker needs to manipulate and load the bitstream to the attacked device by manipulating the non-volatile memory, storing the bitstream, or loading it via some configuration port.
- **Non-Encrypted Attacker Bitstream** The bitstream loaded to the FPGA by the attacker must not be encrypted as the `RDW_GO` command, which triggers the decryption, is replaced.
- **Fuses** Setting the *AES only* fuse, allowing only encrypted bitstreams, prevents the attack. However, enabling the *RSA only* fuse, which solely permits RSA-authenticated bitstreams, does not prevent the attack.

5.2.4 Discovering the Attack by Fuzzing

We found this attack 3 times during our fuzzing experiments by applying different fuzzers. We describe the most straightforward one in the following, shown in Listing 2. With this fuzzer, we inserted 3 writes to the command register at the unauthenticated end of a plaintext RSA bitstream. We chose said register as these commands directly influence the FPGA state and play an immersive part in the startup sequence. The fuzzer is configured to use a different RSA public key than written to the fuses, resulting in an invalid RSA signature in the bitstream. Hence, if the *RSA only* fuse is set, the device should not be able to start. Due to the impact of the command register, often influencing the device state, we narrowed down the crash settings. To this end, we focused on identifying instances where the configuration engine does not report any errors or indicates a successful device startup in the `STAT` or `BOOTSTS` register.

Fuzzer Results The fuzzer processed 32,768 test cases in 17 hours. As stated previously, for valid RSA bitstreams to start, JTAG needs to execute a large number of cycles to process the `RDW_GO` command, increasing the runtime of this fuzzer. Note that we ran the tests on a KCU116 board with the *RSA only* fuse enabled.

Only 2 test cases are logged as crashed, where the device does not show any error, and the done pin is high, indicating a successful start. In both test cases, the sent bitstreams are mostly the same, except for 2 commands that have been rearranged. Investigating these commands reveals that they are part of a regular startup sequence to boot up the device. Usually, such a startup sequence is present in the bitstream footer. In order

to verify the attack, we modified a valid design by invalidating the RSA signature and replacing the `RDW_GO` command with a valid startup sequence.

5.2.5 Extending the Attack

We also tried to extend this attack to break confidentiality, i.e., the encryption. While we tried several fuzzing and rapid prototyping ideas to enable the decryptor, extending the attack remains unsuccessful. Our main idea to extend the attack is to load an ICAP controller with the JustSTART attack, then decrypt a short bitstream and read the decrypted content back via ICAP. The ICAP interface is an internal configuration port considered trusted and thus lacks most security features. Reading back any fabric content via ICAP is allowed even when encrypted bitstreams are loaded.

While it is possible to load such an ICAP controller with our attack, we remain unsuccessful in decrypting a previously encrypted bitstream. The issue is two-fold. First, with fuzzing, we tried to activate the decryption bit in the `CTL0` register, i.e., enable the decryptor after a plaintext RSA bitstream is loaded, which seems impossible. Second, if the decryption bit is enabled, running a decryption using a non-RSA-authenticated encrypted bitstream is impossible, as we confirmed by rapid prototyping. Already the documentation states [Pet21] that if RSA-authenticated bitstreams are used, partial reconfiguration is only allowed via ICAP with unencrypted and unauthenticated bitstreams.

5.3 Investigating Unknown Register 23

In this case study, we investigate the unknown register 23. For that, we used our default intra-register fuzzer and further investigated the crashes by means of rapid prototyping. We uncovered some bits leading to an unresponsive state of the FPGA, which can only be resolved by a power cycle. These experiments were carried out on the UltraScale(+) with an AES key programmed to the BBRAM. 7-Series FPGAs are also sensitive to different bit combinations' nuances.

Intra Register Fuzzer To investigate an unknown register, we choose the intra-register strategy (cf. Section 4) with a broad fuzzing mask by default. This ensures that we get as much information as possible about the unknown register. Since register 23 is unknown, e.g., it is not documented, and there are no bitstreams we could generate containing a write to the register, no individual bits or bit pattern are known. This said we wanted to test most of all possible values the register could hold while respecting the speed limitation of fuzzing hardware. Therefore, we split the 32-bit register into two 18-bit fuzzing masks to test. The two masks overlap in 2 bits, e.g., bit patterns in that region are likely to be caught while reducing the fuzzing time. Hence, this default intra-register fuzzer generates and tests 2^{19} bitstreams.

Investigating the Crashes When executing this default intra-register fuzzer for the unknown register 23 without defining register-specific crash settings, every test case is logged as crashed, and the fuzzer exits after 128 crashed test cases by default. It would not make sense to keep the fuzzer running because if every test case crashes, all test cases need to be analyzed manually, which is unpractical and defeats the purpose of fuzzing. The crashes are rooted in the fact that the fuzzer does not expect any value returned to the register, but in the case of register 23, it returns the written value. Fortunately, this provides the strongest evidence for the presence of that specific register. An additional indication for the existence of a register is if other registers – which are always probed – change. Hence, for the unknown register 23, it is necessary to define crash settings to expect the written value in the register. This starts the process of investigating the crashes,

i.e., iteratively refining the crash settings and continuously learning more about the register and the functionality of the register bits.

Manual Analysis and Testing Hypothesis This iterative analysis includes inspecting the results and comparing the values written to the unknown register 23. Also, we wrote a helper script that searches for identical bits in all transmitted values to identify bits that cause certain types of crashes.

Once we have observed specific bits influencing the state and formed a working hypothesis. We then used the following method to test it. The identified bit is fixated in the fuzzing mask such that only the remaining bits are fuzzed. The crash settings are defined to reflect the presumed hypothesis. Then, if no crash occurs, the working assumption has been confirmed: that the fixed bit has the expected influence on the FPGA state.

Also, we can continue using `ConFuzz` as a rapid prototyping tool to manually investigate found crash cases by setting different static bits and changing the fuzzing masks accordingly. Again, after gaining such information, it is possible to exclude a bit (or bits) from the default fuzzing process to continue fuzzing as initially. Separately, the excluded bit or bit groups can be further investigated manually or with specific fuzzers. Note that it is also possible to use the rapid prototyping feature of `ConFuzz` to encrypt the bitstream or insert additional commands, e.g., writes to the fabric, before or after the write to the investigated register, and observe if this changes the behavior of the target device.

Bit 24 With this method, we first revealed that the FPGA soft crashes if bit 24 is set, e.g., all registers return zero, but the FPGA can be reset by the `JPROGRAM` JTAG command.

Bit 16 and 17 After this new information gain, we removed bit 24 from our fuzzing masks and investigated the remaining bits. Analyzing the newly found crashes reveals an influence of bits 16 and 17 on the data read from the `FDRO` register, which is used to read back the fabric data, i.e., the FPGA design. As per the documentation, it is advised to read an additional frame when accessing the `FDRO` register, along with 10 words (in UltraScale) or 25 words (in UltraScale+) to consider pipelining [Xil23]. If bit 16 is set, single bits are set in the pipeline frame, and if bit 17 is set, the sync word can be seen in the pipeline words. Unfortunately, we cannot comprehensively explain the results, which we will address in future work.

Bit 23, 25, and the Influence of a Programmed BBRAM Key We noticed by coincidence that some of the tested values in register 23 flag the BBRAM key for deletion after a power cycle. In other words, the BBRAM is cleared after testing the unknown register 23 and a power cycle. Running the fuzzer again if *no* BBRAM key is programmed reveals that the FPGA hard crashes if bit 23 and bit 25 are set. Hence, no communication is possible with the FPGA, i.e., it is unresponsive. Only manually power cycling the device resets the FPGA, an error state we only encountered during the investigation of the unknown register 23. Further investigating this hard crash reveals that before the FPGA goes into the unresponsive state, it returns 156 bytes of seemingly random data. Re-executing this crashed test case and comparing the data shows similarities in that data. For example, bits at certain positions are always identical. We assume the returned data are kind of a crash dump in response to the value written to the unknown register 23 before the device locks in an error state.
For a more extensive list of our fuzzing results related to the unknown register 23 (and other registers), we refer to our GitHub repository [Emb23].

6 Discussion & Future Work

The unpatchable nature of FPGA configuration engines underscores the critical importance of proactive security measures during hardware systems design and development. In this work, we highlight the effectiveness and efficiency of hardware fuzzing by automated identification of critical vulnerabilities among other safety and reliability-threatening states, as well as contributing to the general understanding of the configuration engine, thus addressing our research question.

Security Implications With the unpatchable JustSTART attack, we circumvent the RSA bitstream authentication. This vulnerability enables attackers to load non-encrypted bitstreams containing hardware Trojans or manipulate a plaintext RSA-authenticated bitstream, resulting in the leakage of sensitive data during runtime. We want to highlight that this attack can be mitigated by enforcing AES encryption using security fuses, as our attack does not bypass the decryption process. So, JustSTART only breaks authentication, but it does not break confidentiality. In contrast, the starbleed attack targets bitstream confidentiality and is mitigated in case the bitstream is authenticated. Thus, mitigating both attacks simultaneously is only possible by enforcing both **authentication and confidentiality** using the security fuses, as already highlighted in [ELMP22].

FPGA Bitstream Authentication vs. Encryption FPGA bitstream protection schemes typically offer both authenticity and confidentiality as security properties. In various scenarios, authentication-only is a valid security goal as FPGA design confidentiality is not required, e.g., open-source FPGA designs where the HDL is openly available anyways, or when confidentiality is not a functional (user) requirement. For instance, the patchable bitstream encryption engine [UJH+19] requires only authentication for not being manipulated. As another example, the recent DoD report [Age22] thoroughly analyzes potential threats and countermeasures for FPGA assurance. For instance, to prevent adversaries from swapping the bitstream (cf. TD 6), they only recommend authentication. Generally, the report specifically excludes scenarios where confidentiality is required, which are addressed in other reports. We also want to highlight that FPGA security configuration errors by hardware designers may lead to authentication-only bitstream protection due to dispersed and inconsistent official documentation, cf. [AEM+23]. Xilinx does not clearly recommend using authentication *and* encryption, i.e., XAPP1098 [Pet21] discusses the use of *authentication of unencrypted bitstreams*. Based on our research, Xilinx will update its public documentation as part of the vulnerability disclosure process.

`ConFuzz` Framework Our primary motivation for `ConFuzz` was providing a fuzzing framework with rapid prototyping capability to evaluate – security-related – configuration engine hypotheses with an automated workflow. Note that Ender et al. [EMP20, ELMP22] identified the starbleed attack by investigating the documentation and manual bitstream generation. The results discussed in Section 5 show that we automatically rediscover starbleed (even working with other registers than the original attack), and we identified the novel JustSTART attack, resulting in a loss of authentication. Moreover, we want to emphasize that besides the identification of vulnerabilities, a key motivation is to improve the understanding of the configuration engine since the official documentation is incomplete and dispersed. We refer interested readers to [Emb23] for comprehensive evaluation results.

A key advantage of `ConFuzz` is that defining bitstreams in code is considerably less error-prone than manual construction using a hex editor. Note that this drastically reduces required time to construct mutated bitstreams and accelerates hypothesis evaluation.

For example, we leveraged this rapid prototype approach to quickly identify an undocumented RSA test mode mentioned in a Xilinx patent [WAMP13]. We also want to

highlight that the underlying boofuzz framework [Per] provides valuable features and enables fast implementation of necessary extensions to communicate with target devices and craft custom-tailored optimizations.

Systematic Fuzzing Strategies Our investigation into the FPGA configuration engine was guided by a systematic and iterative approach (Section 4), i.e., i) select and ii) implement a fuzzer with the help of our three main strategies, iii) gather, and iv) evaluate FPGA state information to then adjust fuzzer mutation accordingly. Throughout this approach, we were able to establish an improved understanding of the internal workings of the configuration engine, uncovering vulnerabilities and shedding light on various aspects of its functionality. However, we also want to highlight the limitations of such an iterative approach.

i) The approach depends on natural language interpretation of public documentation and patents, often lacking essential technical details such as clear concepts and implementation details. This highlights the critical need for comprehensive documentation in facilitating effective security analysis.

ii) Since this process typically requires guidance by a security engineer, complex fuzzing sequences are not created automatically so far. As of now, this limits our evaluation and mostly sheds light on single registers rather than complex combinations and interactions of them (c.f. next paragraph).

iii) Most of our evaluation assists in understanding the configuration engine's inner workings to a limited extent, i.e., uncovering the behavior vs. purpose. The fuzzing reveals certain inner working effects (i.e., the behavior of the configuration engine). However, it does not help to explain them due to the limited internal status information (i.e., what is the purpose of a particular bit-field?). Therefore, we believe running the current setup for a prolonged period may yield new crashes, but the absence of a comprehensive way to interpret results limits potential for novel insights.

In future work, we plan to address these issues with feedback-based and coverage fuzzing strategies by incorporating FPGA state information as feedback into mutation generation. Such a fuzzer should ideally automatically create new mutations based on feedback and coverage of bit patterns within the bitstream structure. During our evaluation, we observed that adjusting crash settings rather than using a new bitstream mutation strategy yielded more *useful* crash results. Note that this may be due to the fact that we have focused on single configuration register observations so far rather than complex register state interactions. To also address the third limitation, future work should focus on automatic generation of a (detailed) configuration engine state-machine model to enable an improved understanding of its inner workings.

Vulnerability Discovery We now highlight our vulnerability discovery process using our framework `ConFuzz`. Based on rapid prototyping design strategies, we built automated fuzzers to i) run randomized structure-aware fuzzers and ii) validate attack hypotheses by definition of explicit crash settings.
In case of (structure-aware) intra-command and structural fuzzers, we formulated the crash settings as deviating from default initialized values and then let the fuzzer run for a certain time until crashes occurred. In case of the JustSTART vulnerability, we formulated the attack goal that the FPGA boots an authenticated but manipulated bitstream. We then defined this objective as a crash setting (cf. Listing 2) in our fuzzing context and then fuzzed the configuration engine until a crash, i.e., an unauthorized boot-up occurred. Note that in each crash report, `ConFuzz` automatically includes the corresponding bitstream

responsible for said crash. Similarly, we discussed extending JustSTART with (un-)setting the DEC bit in Section 5.2.5, where we narrowed the crash setting only to catch a set or reset of said bit.

From a high-level perspective, we observed that random fuzzing might yield vulnerabilities such as denial of service. However, concrete crash formulation based on a guided idea yielded more effective attacks, as demonstrated in this work.

On HDL Fuzzing While HDL fuzzing may appear more effective and efficient at first glance, confirmation of vulnerabilities (and their absence) post-silicon remains essential even if HDL source code is at hand, particularly considering the difference between high-level HDL and an optimized fabricated chip. To this end, our framework `ConFuzz` can be integrated into a typical CI/CD pipeline. Further, there are gray-box settings – as our current one – where only documentation and a chip without the HDL code are available.

Performance Another limitation is the test-case performance rooted in the connection to devices under test. The utilized JTAG interface uses a single-bit interface, clocked at 66 MHz only. In our experiments, we could test up to 2^{21} bitstreams per day on UltraScale(+) and on our 7-Series cluster 2^{25}. Reasonable enough to run fuzzers discovering undefined behavior and said attacks. However, we are practically limited to fuzzing a complete 32-bit register or sequences of several commands consisting of fewer bits. Note that the test-case performance depends on fuzzer configurations like bitstream length and crash settings.

For future research, one may evaluate an UltraScale(+) FPGA device cluster or use the SelectMAP interface that employs a 32-bit bus and higher speed, theoretically being up to 66 times faster than JTAG. Furthermore, our work can be extended to analyze the JTAG controller itself, as it is used for programming the BBRAM key and security fuses.

Vendors We want to emphasize that our work is centered around Xilinx FPGAs, which share a basic architecture, particularly the configuration process and configuration packets. Using `ConFuzz`, we demonstrated the possibility of rapidly generating and fuzzing bitstreams to discover vulnerabilities. We are confident this concept can be transferred to FPGAs from other vendors, so we plan to expand our work in this regard in the future. However, keep in mind that processor-based systems such as Xilinx Zynq devices have to be handled with a different approach, i.e., processor-based fuzzer approaches such as sandsifter [Dom17].

7 Conclusions

In this work, we demonstrated the effectiveness and efficacy of hardware fuzzing for the Xilinx 7-Series and UltraScale(+) configuration engine with only the chip and limited documentation at hand. We designed and implemented `ConFuzz`, a framework designed to facilitate the creation of structure-aware mutational fuzzers. Additionally, its rapid prototyping capabilities empower analysts to delve deeper into the fuzzing results and swiftly validate assumptions. We formulated three primary fuzzing strategies to establish a systematic approach to fuzz Xilinx FPGA configuration engines. Besides rediscovering the starbleed [EMP20, ELMP22] attack, we identified a new unpatchable vulnerability named *JustSTART* (CVE-2023-20570) that bypasses RSA authentication of all Xilinx UltraScale(+) FPGAs. Moreover, we revealed various undocumented and unexpected behavior, e.g., an RSA test mode or a bit pattern crashing the FPGAs into an unresponsive state.

Since we believe that our work raises awareness for hardware security designers and analysts, we released our fuzzing and rapid prototyping framework `ConFuzz` under the MIT license [Emb23].

Acknowledgments

This work was partly supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA 390781972). We also thank our reviewers for their helpful feedback and Xilinx for their excellent responsible disclosure process, which started on February 27th, 2023, and was acknowledged a day later.

References

[AEM+23] Nils Albartus, Maik Ender, Jan-Niklas Möller, Marc Fyrbiak, Christof Paar, and Russell Tessier. On the Malicious Potential of Xilinx' Internal Configuration Access Port (ICAP). *ACM Trans. Reconfigurable Technol. Syst.*, nov 2023. Just Accepted.

[Age22] National Security Agency. DoD Microelectronics: Field mamable Gate Array Level of Assurance 1 Best Practices, December 2022. https://media.defense.gov/2022/Dec/08/2003127936/-1/-1/0/CTR_DOD_MICROELECTRONICS-FPGA_LOA1_BEST_PRACTICES.PDF.

[AVR14] Vincent Alimi, Sylvain Vernois, and Christophe Rosenberger. Analysis of embedded applications by evolutionary fuzzing. In *International Conference on High Performance Computing & Simulation, HPCS 2014, Bologna, Italy, 21-25 July, 2014*, pages 551–557. IEEE, 2014.

[BHT14] Fabian van den Broek, Brinio Hond, and Arturo Cedillo Torres. Security Testing of GSM Implementations. In Jan Jürjens, Frank Piessens, and Nataliia Bielova, editors, *Engineering Secure Software and Systems - 6th International Symposium, ESSoS 2014, Munich, Germany, February 26-28, 2014, Proceedings*, volume 8364 of *Lecture Notes in Computer Science*, pages 179–195. Springer, 2014.

[CFH+22] Yixuan Cheng, Wenqing Fan, Wei Huang, Gaoqing Yu, Yu Han, Hang Dong, and Wen Liu. PDFuzzerGen: Policy-Driven Black-Box Fuzzer Generation for Smart Devices. *Security and Communication Networks*, 2022:e9788219, 2022. Publisher: Hindawi.

[CRD+23] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. Taylor, M. Egele, and A. Joshi. ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[DCSSK21] Mukta Debnath, Animesh Basak Chowdhury, Debasri Saha, and Susmita Sur-Kolay. FuCE: Fuzzing+Concolic Execution guided Trojan Detection in Synthesizable Hardware Designs. *CoRR*, abs/2111.00805, 2021. arXiv: 2111.00805.

[Dom17] Christopher Domas. Breaking the x86 ISA. *Black Hat*, 2017. https://github.com/xoreaxeaxeax/sandsifter/.

[DWZZ13] Zheng Ding, Qiang Wu, Yizhong Zhang, and Linjie Zhu. Deriving an NCD file from an FPGA bitstream: Methodology, architecture and evaluation. *Microprocessors and Microsystems - Embedded Hardware Design*, 37(3):299–312, 2013.

[EFI21] Maialen Eceiza, Jose Luis Flores, and Mikel Iturbe. Fuzzing the Internet of Things: A Review on the Techniques and Challenges for Efficient Vulnerability Discovery in Embedded Systems. *IEEE Internet Things J.*, 8(13):10390–10411, 2021.

[ELMP22] Maik Ender, Gregor Leander, Amir Moradi, and Christof Paar. A Cautionary Note on Protecting Xilinx' UltraScale(+) Bitstream Encryption and Authentication Engine. In *30th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2022, New York City, NY, USA, May 15-18, 2022*, pages 1–9. IEEE, 2022.

[Emb23] Embedded Security Group. ConFuzz - GitHub. https://github.com/emsec/ConFuzz, 2023.

[EMP20] Maik Ender, Amir Moradi, and Christof Paar. The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1803–1819. USENIX Association, 2020.

[EMS+22] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. Embedded fuzzing: a review of challenges, tools, and solutions. *Cybersecur.*, 5(1):18, 2022.

[ESSG22] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. Rapid Prototyping for Microarchitectural Attacks. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 3861–3877. USENIX Association, 2022.

[ESW+19] Maik Ender, Pawel Swierczynski, Sebastian Wallat, Matthias Wilhelm, Paul Martin Knopp, and Christof Paar. Insights into the Mind of a Trojan Designer: The Challenge to Integrate a Trojan into the Bitstream. In Toshiyuki Shibuya, editor, *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*, pages 112–119. ACM, 2019.

[FS19] F-Secure. CVE-2019-5478. https://github.com/f-secure-foundry/advisories/blob/master/Security_Advisory-Ref_FSC-HWSEC-VR2019-0001-Xilinx_ZU+-Encrypt_Only_Secure_Boot_bypass.txt, 2019.

[HLF+20] Benjamin Hettwer, Sebastien Leger, Daniel Fennes, Stefan Gehrer, and Tim Güneysu. Side-Channel Analysis of the Xilinx Zynq UltraScale+ Encryption Engine. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(1):279–304, Dec. 2020.

[HYJ+17] Andrew Henderson, Heng Yin, Guang Jin, Hao Han, and Hongmei Deng. VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices. In Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis, editors, *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, volume 10453 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2017.

[KCR+10] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak N. Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson,

Hovav Shacham, and Stefan Savage. Experimental Security Analysis of a Modern Automobile. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 447–462. IEEE Computer Society, 2010.

[KL13] Nassima Kamel and Jean-Louis Lanet. Analysis of HTTP Protocol Implementation in Smart Card Embedded Web Server. *International Journal of Information and Network Security (IJINS)*, 2, 2013.

[LCC+15] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. Fuzzing CAN Packets into Automobiles. In Leonard Barolli, Makoto Takizawa, Fatos Xhafa, Tomoya Enokido, and Jong Hyuk Park, editors, *29th IEEE International Conference on Advanced Information Networking and Applications, AINA 2015, Gwangju, South Korea, March 24-27, 2015*, pages 817–821. IEEE Computer Society, 2015.

[LGBD19] Hoang M. Le, Daniel Große, Niklas Bruns, and Rolf Drechsler. Detection of Hardware Trojans in SystemC HLS Designs via Coverage-guided Fuzzing. In Jürgen Teich and Franco Fummi, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, pages 602–605. IEEE, 2019.

[LTK+18] Heiko Lohrke, Shahin Tajik, Thilo Krachenfels, Christian Boit, and Jean-Pierre Seifert. Key Extraction Using Thermal Laser Stimulation A Case Study on Xilinx Ultrascale FPGAs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):573–595, 2018.

[MBKP11] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks: extracting keys from xilinx virtex-ii fpgas. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 111–124. ACM, 2011.

[MRBC21] Damiano Melotti, Maxime Rossi-Bellom, and Andrea Continella. Reversing and Fuzzing the Google Titan M Chip. In *Reversing and Offensive-oriented Trends Symposium*, pages 1–10. ACM, 2021.

[MS16] Amir Moradi and Tobias Schneider. Improved Side-Channel Analysis Attacks on Xilinx Bitstream Encryption of 5, 6, and 7 Series. In François-Xavier Standaert and Elisabeth Oswald, editors, *Constructive Side-Channel Analysis and Secure Design - 7th International Workshop, COSADE 2016, Graz, Austria, April 14-15, 2016, Revised Selected Papers*, volume 9689 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 2016.

[MSK+18] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[Not08] Jean-Baptiste Note. debit. https://github.com/djn3m0/debit/tree/master/altera, January 2008.

[NR08] Jean-Baptiste Note and Éric Rannaud. From the bitstream to the netlist. In Mike Hutton and Paul Chow, editors, *Proceedings of the ACM/SIGDA 16th*

International Symposium on Field Programmable Gate Arrays, FPGA 2008, Monterey, California, USA, February 24-26, 2008, page 264. ACM, 2008.

[PA22] Andrea Pferscher and Bernhard K. Aichernig. Stateful Black-Box Fuzzing of Bluetooth Devices Using Automata Learning. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods - 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24-27, 2022, Proceedings*, volume 13260 of *Lecture Notes in Computer Science*, pages 373–392. Springer, 2022.

[Per] Joshua Pereyda. boofuzz: Network Protocol Fuzzing for Humans — boofuzz 0.4.1 documentation. https://boofuzz.readthedocs.io/en/stable/.

[Pet21] Xilinx Ed Peterson. *XAPP1098: Developing Tamper-Resistant Designs with UltraScale and UltraScale+ FPGAs*, March 2021. v1.4.

[QSD+21] Abdullah Qasem, Paria Shirani, Mourad Debbabi, Lingyu Wang, Bernard Lebel, and Basile L. Agba. Automatic Vulnerability Detection in Embedded Devices and Firmware: Survey and Layered Taxonomies. *ACM Comput. Surv.*, 54(2):25:1–25:42, 2021.

[Rat] Dominic Rath. Open On-Chip Debugger. https://openocd.org/.

[SMOP15] Pawel Swierczynski, Amir Moradi, David Oswald, and Christof Paar. Physical Security Evaluation of the Bitstream Encryption Mechanism of Altera Stratix II and Stratix III FPGAs. *TRETS*, 7(4):34:1–34:23, 2015.

[SW12] Sergei Skorobogatov and Christopher Woods. Breakthrough Silicon Scanning Discovers Backdoor in Military Chip. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2012.

[Sym17] SymbiFlow. Project x-ray. https://github.com/SymbiFlow/prjxray, 2017.

[TLSB17] Shahin Tajik, Heiko Lohrke, Jean-Pierre Seifert, and Christian Boit. On the Power of Optical Contactless Probing: Attacking Bitstream Encryption of FPGAs. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1661–1674. ACM, 2017.

[TSC+21] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing Hardware Like Software. *CoRR*, abs/2102.02308, 2021. arXiv: 2102.02308.

[UJH+19] Florian Unterstein, Nisha Jacob, Neil Hanley, Chongyan Gu, and Johann Heyszl. SCA Secure and Updatable Crypto Engines for FPGA SoC Bitstream Decryption. In Chip-Hong Chang, Ulrich Rührmair, Daniel E. Holcomb, and Patrick Schaumont, editors, *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop, ASHES@CCS 2019, London, UK, November 15, 2019*, pages 43–53. ACM, 2019.

[WAMP13] James D. Wesselkamper, James B. Anderson, Jason J. Moore, and Edward S. Peterson. Programmable integrated circuit with DPA-resistant decryption, 2013.

[Xil23] Xilinx. *UG570 UltraScale Architecture Configuration*, April 2023. v1.17.

[XQZ+21] Hang Xu, Ganyu Qin, Junhu Zhu, Zimian Liu, and Zhiqiang Liu. Framework for State-Aware Virtual Hardware Fuzzing. *Wirel. Commun. Mob. Comput.*, 2021:6698311:1–6698311:14, 2021.

A Fuzzer Listings

```
custom_register_settings = {
    "DEFAULT": {
        "crash_if_differs_from_default": "no",
        "crash_if_equal_to": "F0 0D F0 0D, BE EF BE EF, DE AD C0 DE",
    },
    "register0": {
        # Overwrite the default crash setting from the default_register_settings.ini.
        "crash_if_equal_to": "F0 0D F0 0D, BE EF BE EF, DE AD C0 DE",
    },
    "register3": {
        # The FDRO register should only return zeros because encryption is enabled.
        "crash_if_differs_from_default": "yes",
        "crash_if_equal_to": "",
    },
    "register5": {
        # Overwrite the default crash setting from the default_register_settings.ini.
        "crash_if_not_equal_to": "",
    },
}

starbleed_request = Request(
    name="starbleed_request",
    children=(
        FuzzedBitstream(
            name="starbleed_bitstream",
            file_name="write_fdri_bbram_test_key.bit",
            fuzzing_mask=0xF803E7FF,
            fuzzing_position=FuzzPosition(index_start=284, word_count=40),
        )
    ),
)
```

Listing 1: Excerpt of the source code of the starbleed fuzzer.

```
custom_register_settings = {
    "DEFAULT": {"probe": "no"},
    "register7": {
        "probe": "yes",
        "crash_if_differs_from_default": "no",
        # Only crash if BIT13_DONE_INTERNAL_SIGNAL_STATUS or BIT14_DONE_PIN is set.
        "crash_if_some_bits_in_mask_set": "00 00 C0 00",
    },
    "register22": {
        "probe": "yes",
        "crash_if_differs_from_default": "no",
        # Only crash if just BIT00_STATUS_VALID_0 is set.
        "crash_if_equal_to": "00 00 00 01",
    },
}

plaintext_rsa_bitstream_request = Request(
    name="plaintext_rsa_bitstream_request",
    children=(
        # Disable ConfigFallback in the CTL0 register.
        Type1WritePacket(name="write_to_mask", register_address=6),
        Static(name="mask_value", default_value=b"\x00\x00\x05\x01"),
        Type1WritePacket(name="write_to_ctl0", register_address=5),
        Static(name="ctl0_value", default_value=b"\x00\x00\x05\x01"),
        NOP(3),
        PlaintextRSABlockUltraScale(
            name="plaintext_rsa_block",
            children=(
```

```
                # Original RSA header, except ConfigFallback is disabled in the CTL0 register.
                NOP(),
                Type1WritePacket(name="write_to_mask_1", register_address=6),
                Static(name="mask_value_1", default_value=b"\xFF\xFF\xFF\xFF"),
                Type1WritePacket(name="write_to_ctl0_1", register_address=5),
                Static(name="ctl0_value_1", default_value=b"\x00\x00\x05\x01"),
                Type1WritePacket(name="write_to_mask_2", register_address=6),
                Static(name="mask_value_2", default_value=b"\xFF\xF3\xFF\xFF"),
                Type1WritePacket(name="write_to_ctl1", register_address=24),
                Static(name="ctl1_value", default_value=b"\x00\x00\x00\x00"),
                NOP(8),
                Type1WritePacket(name="write_to_far_1", register_address=1),
                Static(name="far_value_1", default_value=b"\x00\x00\x00\x00"),
                Type1WritePacket(name="write_to_cmd_1", register_address=4),
                Static(name="wcfg_code", default_value=b"\x00\x00\x00\x01"),
                NOP(11),
                # 25 or 26 frames of fabric data for plaintext test mode RSA bitstreams.
                Static(
                    name="fabric_data",
                    default_value=b"\xDE\xAD\xC0\xDE" * CONSTANTS.BOARD_CONSTANTS.FRAME_LENGTH * 25,
                ),
                # Original RSA footer, except ConfigFallback is disabled in the CTL0 register.
                NOP(2),
                Type1WritePacket(name="write_to_cmd_2", register_address=4),
                Static(name="grestore_code", default_value=b"\x00\x00\x00\x0A"),
                NOP(2),
                Type1WritePacket(name="write_to_cmd_3", register_address=4),
                Static(name="dghigh_code", default_value=b"\x00\x00\x00\x03"),
                NOP(20),
                Type1WritePacket(name="write_to_cmd_4", register_address=4),
                Static(name="start_code", default_value=b"\x00\x00\x00\x05"),
                NOP(),
                Type1WritePacket(name="write_to_far_2", register_address=1),
                Static(name="far_value_2", default_value=b"\x07\xFC\x00\x00"),
                Type1WritePacket(name="write_to_mask_3", register_address=6),
                Static(name="mask_value_3", default_value=b"\x00\x00\x05\x01"),
                Type1WritePacket(name="write_to_ctl0_2", register_address=5),
                Static(name="ctl0_value_2", default_value=b"\x00\x00\x05\x01"),
                NOP(2),
                Type1WritePacket(name="write_to_cmd_5", register_address=4),
                Static(name="desync_code", default_value=b"\x00\x00\x00\x0D"),
                NOP(119),
            ),
            children_contain_header_and_footer=True,
            key_file_name="test_key_rsa.nky",
            rsa_private_key_file_name="privateKey_wrong.pem",
            test_mode=True,
            rdw_go=False,
        ),
        Type1WritePacket(name="write_to_cmd_1", register_address=4),
            BitstreamWord(
                name="fuzzed_cmd_value_1", static_bits=0x00000000, fuzzing_mask=0x0000001F,
            ),
            NOP(3),
            Type1WritePacket(name="write_to_cmd_2", register_address=4),
            BitstreamWord(
                name="fuzzed_cmd_value_2", static_bits=0x00000000, fuzzing_mask=0x0000001F,
            ),
            NOP(3),
            Type1WritePacket(name="write_to_cmd_3", register_address=4),
            BitstreamWord(
                name="fuzzed_cmd_value_3", static_bits=0x00000000, fuzzing_mask=0x0000001F,
            ),
        NOP(3),
        Type1WritePacket(name="write_to_cmd", register_address=4),
        Static(name="rdw_go_code", default_value=b"\x00\x00\x00\x16"),
        NOP(3),
    ),
)
```

Listing 2: Excerpt of the source code of the JustSTART fuzzer.