IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2021, No. 4, pp. 412–446. DOI:10.46586/tches.v2021.i4.412-446

LifeLine for FPGA Protection: Obfuscated Cryptography for Real-World Security

Florian Stolz[1], Nils Albartus[1,2], Julian Speith[1,2], Simon Klix[1], Clemens Nasenberg[2], Aiden Gula[3], Marc Fyrbiak[2,4], Christof Paar[1,2], Tim Güneysu[1] and Russell Tessier[3]

[1] Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany
{nils.albartus,florian.stolz,tim.gueneysu}@rub.de
[2] Max Planck Institute for Security and Privacy, Bochum, Germany
christof.paar@mpi-sp.org
[3] University of Massachusetts Amherst, MA, USA
{agula,tessier}@umass.edu
[4] emproof, Bochum, Germany
mfyrbiak@emproof.de

Abstract. Over the last decade attacks have repetitively demonstrated that bitstream protection for SRAM-based FPGAs is a persistent problem without a satisfying solution in practice. Hence, real-world hardware designs are prone to intellectual property infringement and malicious manipulation as they are not adequately protected against reverse-engineering.

In this work, we first review state-of-the-art solutions from industry and academia and demonstrate their ineffectiveness with respect to reverse-engineering and design manipulation. We then describe the design and implementation of novel hardware obfuscation primitives based on the intrinsic structure of FPGAs. Based on our primitives, we design and implement LifeLine, a hardware design protection mechanism for FPGAs using hardware/software co-obfuscated cryptography. We show that LifeLine offers effective protection for a real-world adversary model, requires minimal integration effort for hardware designers, and retrofits to already deployed (and so far vulnerable) systems.

Keywords: FPGA Security · Hardware Obfuscation · Software Obfuscation · Reverse Engineering

1 Introduction

Field Programmable Gate Arrays (FPGAs) combine software flexibility with the performance and energy advantages of hardware solutions and are thus widely adopted in a variety of security and safety-sensitive application domains, including industrial automation, aviation, defense, medical devices, and performance accelerators for machine-learning applications. Even though the post-manufacturing re-programmability of FPGAs provides flexibility, it also opens up a critical attack vector, in particular in settings in which an adversary is able to access the FPGA hardware configuration data (a bitstream). Noticeably, the majority of FPGAs in use today are based on Static Random Access Memory (SRAM) technology, and thus requires *external* non-volatile memory to store bitstreams, which are often accessible by adversaries [EMP20].

In order to safeguard bitstream confidentiality, integrity, and authenticity, the major FPGA vendors introduced bitstream encryption schemes about two decades ago. Despite a seemingly simple cryptographic set-up — the FPGA decrypts the encrypted bitstream upon boot-up — providing a sound security design has been a vexing problem for the

Received: 2021-04-15 Accepted: 2021-06-15 Published: 2021-08-11

FPGA industry. Over the last decade, numerous real-world attacks have been proposed, often based on side-channel leakage or protocol failures (cf. Section 2 and Section 3 for more details). As a result, in many cases, an adversary is able to retrieve cryptographic key material for bitstream encryption schemes with moderate effort. Note that the proprietary nature of the bitstream file format also does not offer any security in practice [Syme]. Even additional external secure hardware elements do not provide sufficient protection, as demonstrated in Section 3.1.

The state of FPGA security has been further weakened by recent advances in bitstream reverse engineering. While the reverse engineering (and optionally manipulation) of an unknown bitstream once was a major undertaking due to the proprietary nature of bitstream file formats, there has been major progress in bitstream reverse-engineering over the past few years [ESW+19, LG97, ZAT06, NR08, BSH12, DWZZ13, PHK17, Not08, Ngu16, Sym18]. An active research community has developed tools to translate proprietary bitstreams back to their gate-level netlist representations. Even though this reverse-engineering can be useful for legitimate purposes (e.g., design analysis), it also enables Intellectual Property (IP) infringement and malicious design alteration such as hardware Trojans [WFSP17].

Due to these security shortcomings, Application Specific Integrated Circuits (ASICs) are sometimes the preferred choice when a higher level of security is desired, since (1) they cannot be (easily) manipulated after manufacturing, and (2) can only be reverse engineered with specialized equipment [FWS+18]. However, ASICs may not be the best choice depending on the use case. In general, an ASIC tape-out is more complex and has a longer time-to-market. For low quantities, ASICs are not economically feasible. Furthermore, the possibility of being able to update the FPGA in-field — especially in security use-cases — is an important advantage. Thus if FPGAs are used in medium or high-security scenarios, they require additional protection.

Goals and Contributions In this work, we introduce a flexible and user-specific line of defense to ensure FPGA bitstream confidentiality, integrity, and authenticity. Our goal is to provide a sound solution against functional reverse-engineering even if an adversary has access to the design netlist. To this end, we first carefully review the assumptions and security properties of previous academic and industrial security schemes and demonstrate how their claimed protection security can be defeated using state-of-the-art adversarial capabilities. We then introduce our hybrid hardware/software co-obfuscation approach that is secure against the aforementioned attacks and requires minimal overhead. Our solution consists of novel primitives, including anti-simulation primitives, hardware self-integrity checks, and effective hardware-software-binding to couple the security advantages of hardware and software domains. Our generic implementation strategies require minimal effort by the hardware designer and can be retrofitted into already deployed systems. In summary, our main contributions are:

- **Generic Hardware Patching Attacks.** We carefully analyze various academic and commercially available protection schemes and – based on our insights – present an arsenal of manipulation strategies that invalidate security properties in real-world adversarial settings. Our strategies focus on automated reverse-engineering and subsequent custom-tailored gate-level netlist manipulation.

- **Novel Hardware Obfuscation Primitives.** We propose several hardware primitives that take advantage of the intrinsic structure of FPGAs: (1) self-integrity checks and bitstream manipulation detection based on bitstream structure information, and (2) covert communication channels based on both partial reconfiguration and deliberately injected crosstalk for obfuscation purposes.

- **LifeLine – Hardware/Software Co-Obfuscation.** We present our hybrid hardware-software solution called LifeLine to protect the confidentiality, integrity, and authenticity of FPGA bitstreams in an almost cryptography-free system. To this end, we detail how novel hardware security primitives are coupled with custom-tailored software obfuscation for the embedded domain to yield a dynamic assembly of the final hardware design. In particular, we demonstrate how to leverage software-based point-obfuscation in combination with anti-simulation primitives, such as partial reconfiguration, to yield a strong and efficient obfuscation.
- **Comprehensive Evaluation.** We design and evaluate an obfuscation scheme to realize FPGA bitstream security and an IP core license model for currently available and already-deployed FPGA systems. We demonstrate that our schemes are efficient in terms of overhead and provide effective protection with respect to state-of-the-art obfuscation quality criteria.

2 Technical Background on FPGA Security

Bitstream Encryption FPGAs are digital hardware devices that are designed to be user-programmable after manufacturing. For market-dominating SRAM-based FPGAs, an external non-volatile memory is required to store a user configuration (*bitstream*) that programs the FPGA upon boot-up. Effectively, the bitstream represents a proprietary, encoded gate-level netlist. To counteract IP theft and reverse engineering, FPGA vendors have implemented a variety of bitstream encryption schemes. However, it has been shown that FPGA devices from families across vendors are susceptible to side-channel attacks [MBKP11, MKP12, MOPS13, MS16, SW12, SMOP15, KHPC19] and, as recently demonstrated, to protocol-level attacks [EMP20]. Hence for vulnerable FPGA families, bitstream encryption schemes alone do not provide sufficient protection as an attacker is able to obtain the decrypted bitstreams.

Bitstream Reverse Engineering and Manipulation Even though FPGA vendors typically keep bitstream file formats proprietary, numerous works have demonstrated how to reverse engineer (parts of) a bitstream file-format [ESW+19, LG97, ZAT06, NR08, BSH12, DWZZ13, PHK17, Not08, Ngu16, Sym18]. To automate the tedious step of file-format reverse engineering, hardware designs with *small* differences are typically generated and bit differences in resulting bitstreams are observed.

Figure 1: Netlist Extraction for FPGAs.

For example, SymbiFlow [Sym18] is an open-source Electronic Design Automation (EDA) tool-flow for FPGAs. Projects affiliated with SymbiFlow, such as *IceStorm* (Lattice iCE40) [Symb], *Trellis* (Lattice ECP5) [Symc], *X-Ray* (Xilinx Artix-7, Kintex-7, Zynq-7) [Syme] and *U-Ray* (Xilinx UltraScale(+)) [Symd] aim to document proprietary bitstream file formats. The overall goal of project SymbiFlow is to have a complete open-source tool-flow from Register Transfer Level (RTL) to bitstream. However, based on the documented bitstream file format, a bitstream can be translated back into a gate-level netlist. More precisely, the project provides scripts to translate a given bitstream back to a gate-level netlist including a constraints file with placement and routing information. Hence, reverse engineering and manipulation of hardware designs can be conducted.

Gate-level Netlist Reverse Engineering A gate-level netlist is a low-level hardware design representation that is comprised of a collection of gates and their interconnections [WH15]. In particular, a flattened netlist translated back from a bitstream contains no high-level information about functional modules or any form of hierarchies [AHT+20].

To recover high-level (register-transfer) information from an unstructured gate-level netlist, a reverse-engineer must use (1) control path analysis and (2) data-path analysis. Control path information can be extracted with the identification and extraction of Finite State Machines (FSMs) [STGR10, MZJ16, McE01, MJTZ16, BBS19, AHT+20]. First the FSM candidate circuits are identified and then the state transition graph is reconstructed. To recover data-path information, word-level structures are identified using sub-circuit identification and matching [LGS+13, STP+13], or topology analyses of the underlying graph [FWR+20, AHT+20].

Selected Implementation Attacks on FPGAs Successful third-party FPGA manipulation attacks have been demonstrated. Swierczynski [SFKP15] *et al.* demonstrated an attack on an FPGA-based USB flash drive by manipulating an AES circuit to weaken its cryptographic properties. Moreover, Swierczynski *et al.* [SMOP15] introduced bitstream fault injections that leak key material of cryptographic implementations even if bitstream encryption is enabled. Moraitis *et al.* [MD20] demonstrated the extraction of *Snow 3G* keys by modifying Look-Up Table (LUT) configurations. Most recently Kataria *et al.* [KHPC19] demonstrated the ability to bypass the security of a Cisco Trust Anchor using targeted direct bitstream manipulation.

3 (In-)Effectiveness of State-of-the-Art Solutions

In this section, we discuss industry and academic solutions that attempt to overcome the security weaknesses presented in Section 2.

3.1 Industrial Solution for FPGA Authentication

Commercial security solutions generally provide limited protection for FPGA designs. To this end, we analyze the solution from a large security vendor[1]. The key idea of this security solution is to prevent counterfeit IP (*e.g.*, a valid bitstream cannot be applied to a second FPGA of the same model). The solution is based on an external hardware chip, an *authenticator*, shown in Figure 2. Both the authenticator chip and the FPGA store cryptographic values which are processed in an authentication step. The authenticator solution is promoted for use-cases in medium-security settings, including FPGAs that do not support bitstream encryption.

Reference Implementation Figure 2 shows the design and implementation. The centerpiece of the solution is a challenge-response protocol between an FPGA IP core (*Hash Recipient*) and the authenticator Integrated Circuit (IC) (*Hash Originator*). Both the FPGA and the authenticator compute a value based on a secret. If and only if these values match, the FPGA is authenticated and then activated. Value computation leverages a hash function, such as SHA-256, that takes the secret, device data, and a random challenge as input. The authentication status is determined by the comparison circuit using the authenticator IC hash value and the FPGA output value from the SHA-256, yielding a 1-bit signal that indicates if both values are equal.

[1] Please note that after consultation with the vendor, we agreed to leave out specific product details and names, due to security concerns raised by the vendor.

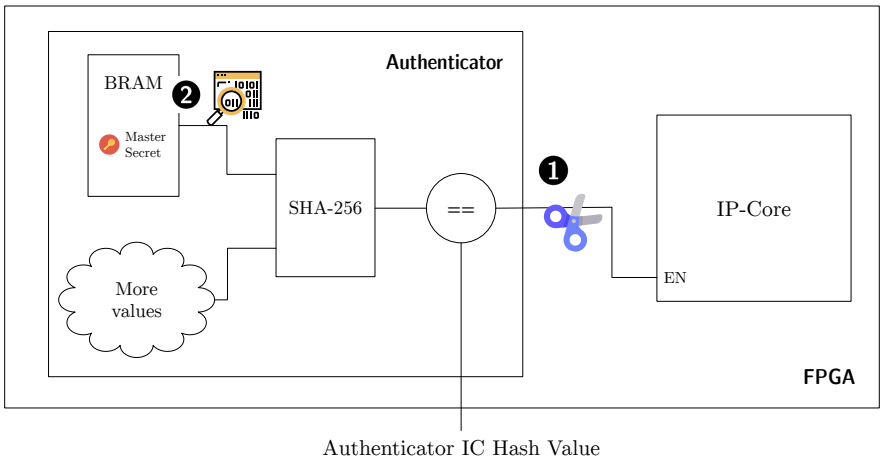


Figure 2: Overview on the FPGA Authentication using an Authenticator IC.

High-Level Attack The vendor indicates that bitstream file format (and subsequent gate-level netlist) reverse-engineering is indeed possible, although due to the obscurity, complexity, and size of the bitstream the process is claimed to be *difficult and time consuming*. However, this type of reverse engineering is now more feasible than it was six years ago (cf. Section 2). Bitstream translation to a gate-level netlist is an automated task that can already be achieved for several FPGA families. We now demonstrate how to invalidate the system's security using two distinct attacks that leverage automated gate-level netlist reverse-engineering and manipulation for Xilinx 7-Series devices, see Figure 2.

A detailed description of the official implementation is only available under Non-Disclosure Agreement (NDA), so we re-implemented the solution to be close to what is publicly described in marketing information, application notes, and technical documents. To this end, we used an open-source SHA-256 core [sec] and implemented a results comparator in Verilog that compares the hash function output to an external value.

❶ Result Comparison Patching As noted earlier, the authentication process ends at a comparison circuit implemented on the FPGA. Hence, we manipulate this circuit so that it will always report *true* irregardless of the available input. In order to automatically identify the circuit's comparators in the gate-level netlist, we exploit their unique inherent structure that allows for automated identification, as proposed by Fyrbiak *et al.* [FWS+18]. For this purpose, we first located comparator circuits in the gate-level netlist, computed their Boolean functions, and then performed formal equivalence checks against a predefined comparator model using the z3 theorem prover. Specifically, we searched for comparators exhibiting the distinct output size of 256 bits. After identification, the respective comparator circuit output net in the bitstream is manipulated to generate a fixed value. An attacker could also manipulate LUTs utilized in the comparator or re-route the source of the enable signal for the IP core to global VCC or a register that always holds `'1'`, however, even small manual changes to the routing can be complex to implement.

We implemented this attack on a Basys3 board that features a Xilinx Artix-7 35T FPGA. To translate the bitstream back to its gate-level netlist, we used the `fasm2bels` tool [Syma]. We then implemented a plugin for HAL [HAL] to automatically identify the aforementioned comparator circuit. Once the entry point for our manipulation has been identified, we attack the programmable interconnect in the bitstream by *cutting* the wire. The user-programmable routing on Xilinx 7-series FPGAs uses *switch matrices* containing numerous Programmable Interconnect Points (PIPs) that are bitstream configurable.

Hence, disabling an active PIP directly translates to cutting the corresponding net within the netlist. In our experiments on Xilinx 7-Series FPGAs, we verified that most PIPs, if unconnected, output a logical `'1'`. Therefore, by cutting the net from the comparator, the comparator output signal is fixed to a constant `'1'` and the hash value check is bypassed entirely. For the final step, we developed a tool based on Project X-Ray [Syme] that reads a bitstream, manipulates its configuration (including the PIPs), and outputs the modified bitstream. After the design manipulation, our tool updates the Cyclic Redundancy Check (CRC) and other Error Correction Code (ECC) values in the bitstream.

❷ **BRAM Secret Extraction** As shown in Figure 2, the shared master secret is "obfuscated in the bit file memory" and thus its confidentiality is based upon the proprietary nature of the bitstream file format. More precisely, the vendor states that the shared secret is stored in Block-RAM (BRAM) and recommends that in *the final design, it is best to change the memory-mapped address so a hacker will not know the actual location in memory.* Since the initial BRAM content is also stored within the bitstream, an attacker can explore the memory in the gate-level netlist in a straightforward manner. Note that the vendor does not mention any additional obfuscation measures to securely store the secret, but merely provides guidance on how to introduce a new 256-bit shared secret by updating the initial BRAM content using Xilinx tooling.

3.2 Vulnerabilities in Academic Solutions

To implement security, most systems require secure key storage to store root-of-trust secret keys. To this end, most academic research focuses on (1) hard-coded device keys [MSV12, VMK+15], (2) Physical Unclonable Functions (PUFs) [GKST07, USZ+20], or (3) obfuscation [HAN18, ZLL+13]. However, in our real-world adversary model virtually all proposed solutions can be circumvented in a straightforward and mostly automated manner as described in the following.

Hard-Coded Device Keys Schemes developed by, for example, Maes *et al.* [MSV12] use a licensing scheme based on a fixed decryption key. It is embedded into a bitstream encrypted with a device-key and provided by a Trusted Third Party (TTP), so it is unknown to the end-user. However, reverse engineering tools such as `DANA` [AHT+20] can be used to identify the key register once the bitstream encryption is broken. Dynamic analysis and netlist patching adds another way to leak the key by rerouting the output to an attacker controller port [WFSP17].

Physical Unclonable Function In these solutions, weak PUFs are used to form a unique device key that provides an input for unlocking mechanisms. PUFs thus represent secure key storage. Static analysis and simulation are unable to identify weak PUFs, however, they are susceptible to dynamic analysis. By probing the signals on the FPGA via netlist patching, the attacker can learn the challenge-response pair. Most solutions use only one or a restricted amount of challenge-response pairs. Therefore, the attacker can patch out the PUF by rerouting the input of the unlocking logic. This patch can either be a static register or an attacker's own logic, which provides an expected response.

Obfuscation FSM obfuscation, *e.g.*, by appending dummy power-up states [HAN18], and Logic Locking, *e.g.*, using a PUF output to mask core logic [ZLL+13], are flawed as shown by Fyrbiak *et al.* [FWD+18] and Engels *et al.* [EHP19]. Thus, binding the device identifier to a unlocking mechanism is ineffective and can be reversed and patched out using automatic tool such as `HAL` cf. Section 2.

4 Takeaways and Security Considerations

Based on previous approaches from industry (cf. Section 3.1) and academia (cf. Section 3.2), it is evident that the use of cryptography alone does not provide a satisfactory level of security in real-world systems. In particular, FPGA applications are faced with the challenge that (vulnerable) bitstream encryption schemes cannot be updated once deployed in FPGA platforms. Moreover, the proprietary nature of bitstream file formats is not an effective measure to prevent reverse-engineering and manipulation. Hence, targeted malicious design manipulation (even for a single wire, cf. Figure 2) can collapse system security.

Design Considerations From a security engineering point of view, we require a mechanism that is able to load an encrypted bitstream onto the FPGA using a computational black box that an adversary cannot inspect. Moreover, we require a solution that retrofits to vulnerable in-field FPGAs. Thus, the approach cannot add custom security hardware, *e.g.*, secure elements or Hardware Security Modules (HSMs), in the FPGA control-plane. This implies that a trust-anchor must be implemented in the (potentially attacker-controlled) FPGA design itself.

Hence, a sound obfuscation scheme is our last line of defense for many (commercial) applications as it meets the aforementioned requirements and avoids the weaknesses of established but limited solutions. Even though many hardware obfuscation schemes exist, they are not sufficient in our setting where we need a strong trust anchor. A key idea of LifeLine is that a combined hardware-software co-obfuscation results in a system with a dramatically higher security level than pure hardware or pure software obfuscation provides in isolation.

Research Question Several hardware obfuscation primitives, including FSM-based obfuscation [FWD+18] and opaque predicates [HP18], are available. Our research explores which obfuscation primitives must be used and how they can be integrated into a complete system to ensure sound protection capabilities within a realistic adversary model. To address the research question, we first introduce several novel hardware obfuscation primitives in Section 5 and combine them with state-of-the-art software obfuscation primitives in Section 6 to leverage the advantages of both worlds.

4.1 Attacker Capabilities

The goal of the adversary is to invalidate the confidentiality, integrity, or authenticity of the FPGA design (e.g., motivated by IP infringement, competitive design analysis, or malicious design manipulation). Analogous to prior research, we assume that the adversary is capable of recovering a perfect, error-free *flattened* netlist (meaning no hierarchy information is available) from the bitstream. We also assume that the adversary can manipulate the netlist (e.g., adding an on-chip debugger to leak run-time hardware information). The adversary has a variety of capabilities at hand, as explained in the following.

Static Analysis Static analysis examines design functionality at the netlist level without executing the design. The netlist is often interpreted as a directed graph which enables the usage of graph theory. The reverse engineer can employ detection algorithms to, *e.g.* identify the control path or perform data-path analysis (cf. Section 2). Tools such as the open-source framework `HAL` [HAL] can be employed for static gate-level netlist analysis.

Dynamic Analysis (Simulation) A reverse engineer can observe the behavior of the chip over time using dynamic analysis. Types of dynamic analysis include *simulation-based* and *on-chip*. Due to the sheer size of a netlist, dynamic analysis cannot usually be carried out without static analysis since the points of interest that the reverse engineer wants to analyze must be identified first. A common tool that can be used to simulate the design is the open-source simulator *Verilator* [Ver]. To simulate a given netlist, the designer needs the correct functionality description of each gate, which is defined by the gate library.

Dynamic Analysis (On-Chip) If an attacker is capable of analyzing on-chip behavior, the execution environment can be observed. This approach allows values processed on the FPGA (*e.g.* cryptographic keys) to leak. Furthermore, the reverse engineer can *e.g.*, observe the behavior of a state machine or deduce data flow on the chip. To conduct on-chip analysis, an attacker must insert a logic analyzer into the bitstream, which can be a challenging task.

Design Manipulation To analyze system behavior, the attacker can perform manipulation of the target design (e.g., patch a self-test and observe whether the design still performs its operation). In particular, the behavior of manipulated and non-manipulated designs can be combined with information from static and dynamic analysis.

4.2 System Overview

We now provide a high-level overview of our complete system to guide the reader through Section 5 and Section 6. To develop a sound obfuscation scheme that withstands state-of-the-art reverse-engineering (and manipulation), we build our solution on two simplified key observations: (1) data-flow in hardware is static and it is challenging for the reverse-engineer to identify the control-flow, (2) the control-flow in software is static and it is challenging for the reverse-engineer to identify the data-flow[2].

We design our system with a dynamic data-flow in hardware (e.g., via partial reconfiguration) and a dynamic control-flow in software (by building on well-established software obfuscation methods). Moreover, we aim to bind hardware control-flow and software data-flow via cryptography (e.g., a software-based hash function that depends on hardware values). In particular, the latter step is crucial for our system security to force an attacker to reverse-engineer both hardware and software where each is armed with custom-tailored obfuscation (e.g., anti-simulation primitives in hardware). In addition, we equip our system with integrity checks to detect hardware and software manipulations. With this approach, we achieve a deep coupling of hardware and software primitives, which creates the foundation for strong hardware/software co-obfuscation.

A Cryptographer's Perspective From a high-level perspective, our system is based on the decryption of a payload bitstream that makes use of a strongly obfuscated decryption engine in software that is interwoven with hardware features, cf. Section 7. The payload bitstream is decrypted on the FPGA system and configured with the help of partial reconfiguration. Coupling of our software and hardware is, among others, established through the cryptographic hash-based key derivation used for the bitstream decryption that is dependent on both obfuscated values in software and hardware. Note that this forces a reverse engineer to analyse both the hardware obfuscation primitives and obfuscated software as a whole.

[2]We acknowledge that the statements about hardware and software systems and reverse-engineering views do not hold for all applications, but we opted for a simplistic view for motivational purposes.

5 FPGA Obfuscation and Security Primitives

We now introduce the design and implementation of our hardware obfuscation primitives based on partial reconfiguration (Section 5.1), covert data-flow based on partial reconfiguration (Section 5.2), crosstalk (Section 5.3), and bitstream self-integrity checks (Section 5.4). From a high-level point of view, each hardware obfuscation method aims to defend against static analysis, dynamic analysis, and hardware design manipulation, respectively.

Table 1: Overview of Hardware Obfuscation Primitives.

Technique	Anti-Static Analysis	Anti-Simulation	Anti-On-Chip Debugging	Integrity
Partial Reconfiguration	●	◒†	○	○
Covert Channel	●	◒†	○	○
Crosstalk	●	●	●	○
`BIMAD`	○	○	●*	●*
LifeLine	●	●	●	●*

† – Anti-simulation protection via partial configuration can only be guaranteed for Xilinx devices.

* – Integrity/on-chip debugging prevention only for user pre-defined routes.

5.1 Partial Reconfiguration

To increase FPGAs hardware resource utilization and transform FPGAs into self-adaptable and more flexible systems, partial reconfiguration is used to dynamically reconfigure parts of the FPGA at run-time. Examples include demand-driven instruction set modification and secure updates of cryptographic algorithms in FPGA-based secure elements [VF18]. Even though tools support partial reconfiguration in hardware designs, the simulation of partial reconfiguration is often not supported by vendors (i.e., Xilinx does not provide simulation features for partial reconfiguration [Xil21]).

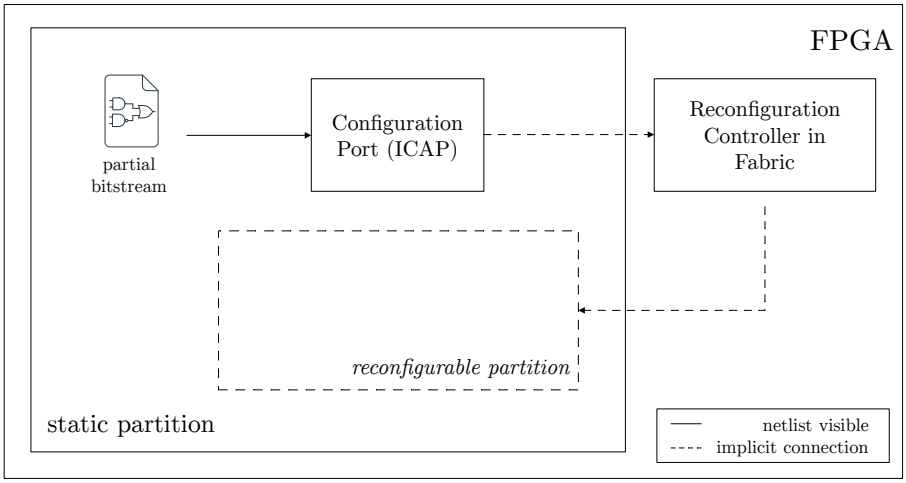


Figure 3: Partial Reconfiguration Overview.

Security Consideration We leverage partial reconfiguration as an anti-static analysis hardware obfuscation primitive. If partial reconfiguration simulation capabilities are not available for an FPGA (*e.g.*, for Xilinx FPGAs), partial reconfiguration also forms an anti-simulation primitive analogous to Fyrbiak *et al.* [FWD+18]. The key idea is to transform hardware resources that handle sensitive data into a partial design. Module configuration of these resources is performed at run-time. Hence an attacker only receives an *incomplete* static gate-level netlist and a partial gate-level netlist.

5.1.1 Design

To leverage the security properties of partial reconfiguration for hardware obfuscation, we use a standard design flow that consists of ❶ partitioning of the hardware design and then ❷ extension of the hardware design with a partial reconfiguration controller to manage dynamic reconfiguration.

❶ **Hardware Design Partitioning** In general, partial reconfiguration requires the partitioning of a hardware design into a static partition and one or more reconfigurable partitions. To this end, the hardware designer performs a *floorplanning step* post-synthesis in which physical resources are allocated for each reconfigurable partition including their physical location on the FPGA. From the viewpoint of the static partition, modules in the reconfigurable partition(s) are *computational black boxes* and translated to partial bitstreams independently. Note that a partial bitstream only represents configuration bits for the reconfigurable partition, so if a module only consists of 10% of FPGA resources, its partial bitstream is approx. 10% the size of a complete bitstream [VF18].

❷ **Partial Reconfiguration Controller** To configure a partial bitstream into its respective partition, the static design is extended to include a partial reconfiguration controller. In particular, the partial reconfiguration controller handles communication with the hardware (re-)configuration engine and manages which partial bitstream is configured into the reconfigurable partition at a specific point in time. Depending on the reconfigurable partition size (and application-specific hardware overhead requirements), the partial bitstreams can be stored either on the FPGA (e.g., in BRAM) or loaded at run-time from external resources (e.g., external flash or even via network).

5.1.2 Implementation

Since hardware design partitioning can be handled as a standard floorplanning step in the EDA toolchain, we refer the interested reader to the respective Xilinx User Guide [Xil21]. We now detail our implementation of the partial reconfiguration controller. From a high-level perspective, the controller consists of an FSM that handles communication with the FPGA family-specific configuration port.

In this work, we leverage a Zynq-7000 FPGA (ZedBoard) as an implementation and evaluation platform. Its ICAPE2 controller [Xil18], which handles dynamic writes to the FPGA bitstream configuration port, is instantiated. Other Xilinx architectures offer additional configuration ports. The partial bitstream header contains all required commands for the ICAPE2 interface to perform configuration. The ICAP is restricted to a maximum frequency of 100 MHz, limiting data throughput in certain applications. Optimized controllers (e.g., based on Direct Memory Access (DMA) [LPF10]) have been developed to saturate ICAP capabilities.

5.2 Covert Communication with Partial Reconfiguration

In this subsection we describe how partial reconfiguration can be used to create a covert communication channel. This approach hinders static and simulation analysis for Xilinx FPGAs. The high-level idea is to populate parts of the datapath via partial reconfiguration. We take advantage of the fact that Flip-Flops (FFs) can be preset by the designer to specific values (`INIT` values) as a result of reconfiguration. This approach allows us to write arbitrary values to registers, without revealing a source within the netlist.

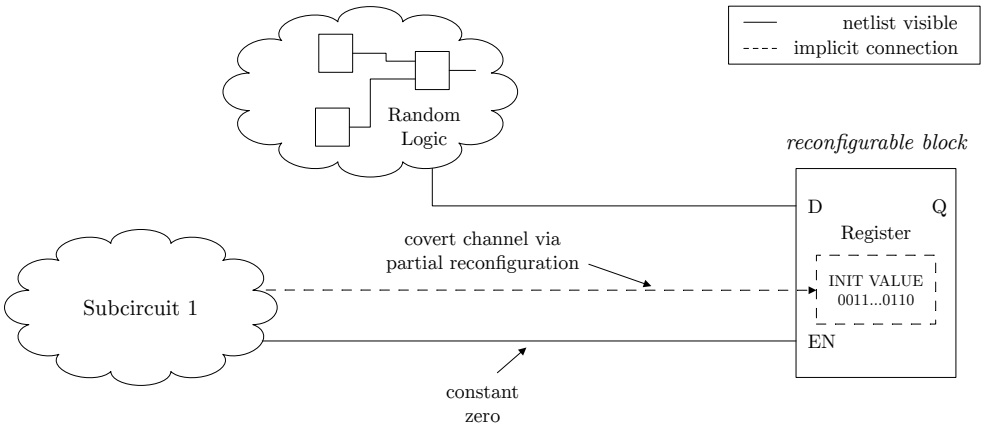


Figure 4: High-Level Overview of Covert Communication with Partial Reconfiguration.

Security Consideration Since the data source connection for the preset value is not visible in the netlist, the reverse engineer cannot conduct static analysis, breaking many automated identification algorithms and tools, such as `DANA` [AHT+20], due to false and missing connections. Not only is the connection not visible in the netlist, but the FF can be connected to a dummy input making the re-configurable register look like it is driven from different logic. This makes identification of the data channel nearly impossible.

5.2.1 Design

To realize the covert channel, we use the partial reconfiguration controller, as described in Section 5.1.1. Figure 5 shows the setup for covert communication. Whenever the subcircuit wants to transmit specific data, it issues a send command to the RISC-V softcore (❶). Depending on the data to transmit, the software selects the partial bitstream which contains the specific `INIT` values for the register the subcircuit wants to write (❷). To transmit 4 bits, as depicted in the workflow, 16 partial bitstreams are needed to realize all possible communications (from `0000`, `0001`, ... `1111`). Once the correct partial bitstream has been selected, it is sent to the reconfiguration controller (❸), which in turn reconfigures the register in the dynamic partition (❹). In this case, the reconfiguration controller is a RISC-V softcore. The value to be transmitted from *subcircuit 1* is written to a memory mapped register. The core handles the partial reconfiguration by communicating with the reconfiguration port.

Since the register D input is connected to *random* logic, the designer must ensure that the register enable *EN* is never set to '1'. This goal can be achieved by connecting the enable to one of the memory mapped registers of the softcore that is never set to '1'. This configuration disguises the connection to look *real* to the reverse engineer, while at the same time ensuring that the intended value of the register is not overwritten. The setup

leaves almost no chance for the reverse engineer to figure out the correct connection with static analysis or simulation-based analysis for Xilinx FPGAs.

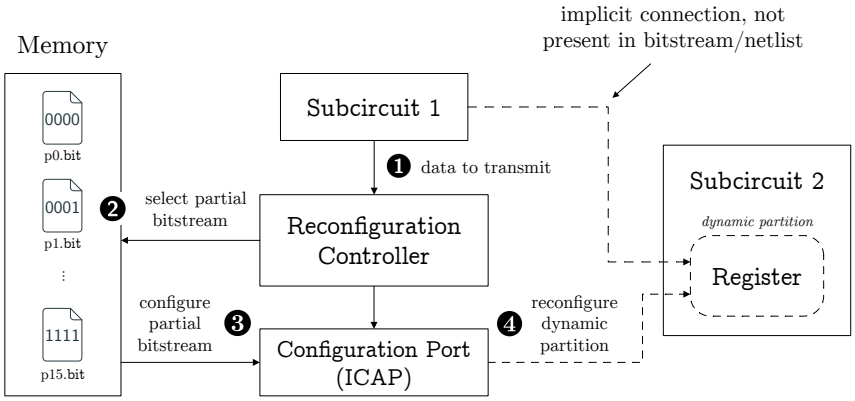


Figure 5: Setup for Covert Communication with Partial Reconfiguration.

5.2.2 Implementation

We implemented the register as a simple Verilog module which contains a flip-flop holding a specific initial value. In the netlist, it appears that the register value may be overwritten by input from random logic. To ensure that the Xilinx toolchain does not optimize the register away, a `DONT_TOUCH` attribute is attached to the register to avoid circuit changes during synthesis. The `RESET_AFTER_RECONFIG` flag for the corresponding reconfigurable partition is also used to reset the register to its initial value after reconfiguration.

On the Zynq-7000 platform, each partial bitstream consumes about 50 kBytes on average with bitstream compression enabled. On platforms with more flexible partial reconfiguration, such as UltraScale devices, it is possible to create even smaller partial bitstreams, as reconfigurable regions (pBlocks) must not obey strict size rules [Xil21].

Improvements In the current implementation it is necessary to store 2^n partial bitstreams for data transmission of n bits. This issue could be overcome by storing a single partial bitstream for the register in addition to the FF position and `INIT` value in the bitstream. The reconfiguration controller would need to adjust the error correction bits in the associated frame for each FF where the `INIT` value is changed, and the CRC checksum for the partial bitstream. The corresponding bitstream positions in Xilinx 7-Series devices can be found in the Project X-Ray database.

5.3 Crosstalk

It has recently been shown that neighboring wires in an FPGA routing channel influence each other's delay in measurable ways [RPD+18, GRE18]. This signal crosstalk[3] allows a covert communication channel to be formed between adjacent wires in a routing channel that is not physically connected to each other. Recent work [RPD+18, GRE18] has focused on an attacker using crosstalk to snoop on an adjacent wire of a victim if multiple independent users simultaneously share an FPGA substrate. In this work, we use adjacent-wire FPGA crosstalk as a covert communication channel within a single user's design in a deliberate effort to further obfuscate circuit function from a potential attacker with access to the design netlist.

[3]The term "crosstalk" often refers specifically to capacitive coupling between wires. In this paper we use the word crosstalk in a more general sense to describe the unspecified interaction between neighboring wires.

Security Considerations Effectively, we use crosstalk between two adjacent FPGA wires as a primitive to transmit values between a transmitter/receiver pair without a physical connection. The implementation mitigates the risk of functional reverse engineering from a netlist by hiding communication between sub-circuits (cf. Figure 6). To identify possible crosstalk communication, an attacker needs logic synthesis, placement, and routing information. The attacker also needs to know the routing channel layout of the device. The information needed to locate and analyze this data far exceeds what is needed to simply analyze the gate-level netlist.

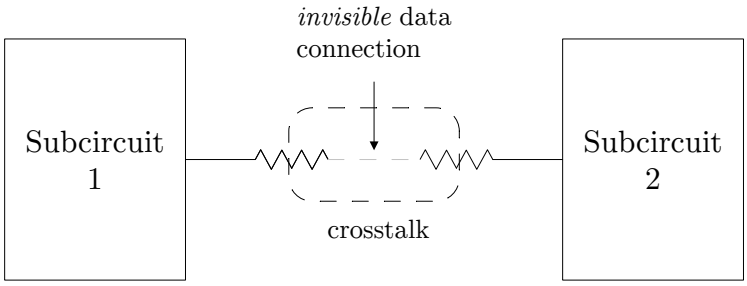


Figure 6: High-Level Idea of Crosstalk Communication.

A crosstalk obfuscation primitive protects against both static and simulation analysis, and partially against on-chip debugging. An attacker cannot correctly simulate or statically analyze the netlist since the adjacent-wire communication channel connection is not represented in the netlist and the datapath connection is only present on the device itself. Since crosstalk parameters vary across FPGA families, an attacker would need to analyze the design physically on a specific device to understand what type of communication is taking place.

5.3.1 Implementation

To detect the logic state of an adjacent wire, an asynchronous Ring Oscillator (RO) [GRE18] is used as the receiver. The ring oscillator's frequency is affected by the state of the adjacent transmitter wire. The oscillator frequency is greater when an adjacent wire holds a high voltage and vice versa when the wire is low. The preceding characteristic allows the receiver to make a guess of the transmitter state with high probability over some predetermined sampling period (T). The magnitude of this effect is dependent upon the device family, length of the wire overlap, technology node, and is partly influenced by the unique silicon variation of the device. A binary counter is connected to a RO output to determine how many oscillations take place during the sampling period. Higher counts (reduced RO delay) indicate the presence of a static logic `'1'` on the adjacent wire.

Figure 7 demonstrates the crosstalk effect and its influence on the RO count depending on the logic state of the transmitter on a Zynq 7000 device. Each of the 120 experiments resulted in a count obtained after $T = 2.1$ ms. First, an experiment was performed with a transmitted logic `'1'` (blue dot) followed by an experiment with a logic `'0'` (red dot). This procedure was repeated for the rest of the measurements. Counts obtained for each experiment are shown in the figure. The channel wires overlap for a span of 140 logic slices. The black line shows a classification point that can be used by the receiver to guess the state of the transmitted data. Since there is no 1-D line that classifies every possible point correctly, the transmission accuracy is not 100% for all samples. Therefore, error detection is utilized to minimize errors on the receiving side.

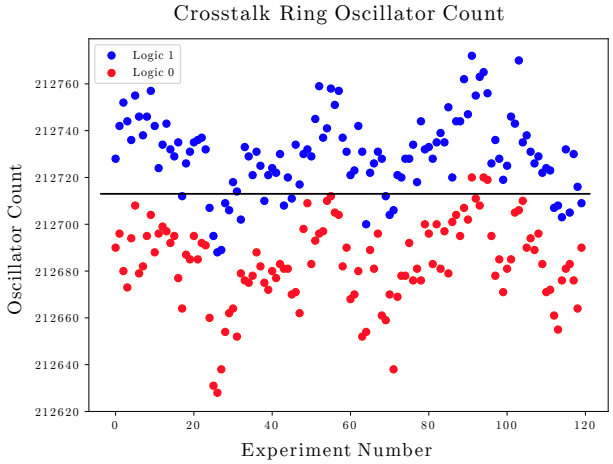


Figure 7: Ring Oscillator Count with 120 Experiments Performed.

To use crosstalk for data transfer, the design is split into two parts that do not have physical connections. One part includes the transmitter, which encodes the data while simultaneously shifting it out onto a transmitter wire at a specified frequency. The other module includes the receiver, which monitors the effective frequency of the RO and dynamically decides what logic state has been transmitted. The data is then decoded and determined to be valid or invalid based on error checking. To successfully send data, the transmitter and receiver are synchronized at the same data throughput rate. Figure 8 demonstrates the architecture for crosstalk-based data transmission.

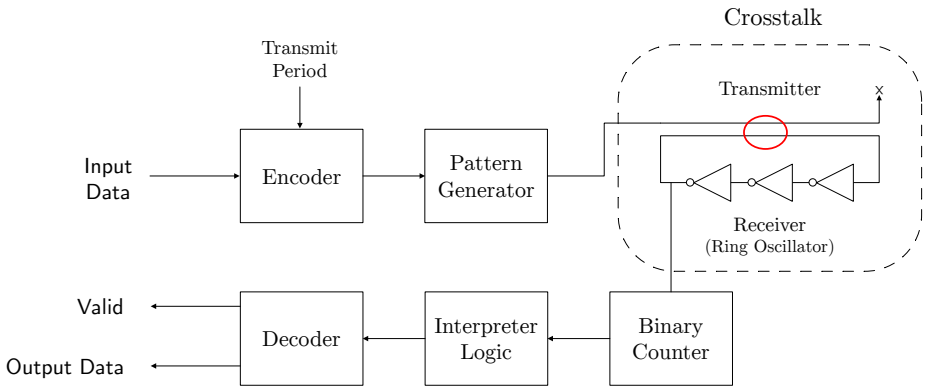
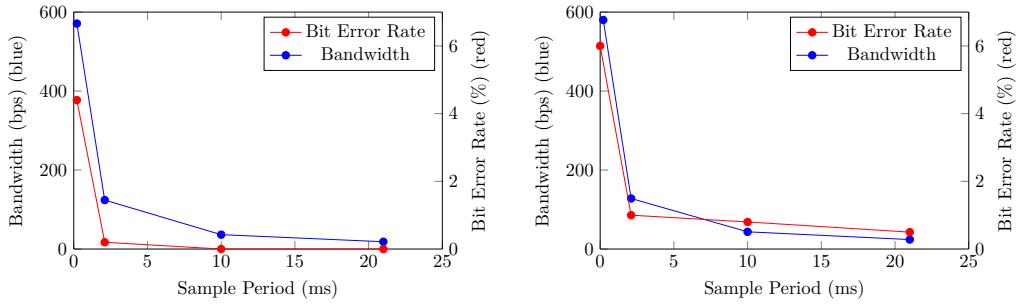


Figure 8: Crosstalk Primitive Overview.

The transmitter wire and receiver wire overlap in a routing channel is achieved by using directed routing constraints and logic isolation to reduce noise during FPGA design compilation. The length of this overlap linearly influences the intensity of the crosstalk effect. The encoder module employs an 8-bit CRC per chunk of transmitted data (*e.g.*, 16 bits) for error detection. The pattern generator module includes a shift register that is controlled by the transmit period. The binary counter measures the effective frequency of the RO. The interpreter module uses the synchronized counter value to determine the logic state of the transmitter. At the start of each transmission request, a configuration sequence is used to synchronize the interpreter. The decoder then accumulates all values and validates data integrity based on the code word that comes through the channel. If the codeword is invalid, the controlling entity (Section 7) must initiate a new transmission.

5.3.2 Evaluation

In this subsection, we evaluate the bandwidth in bits per second (bps) and Bit Error Rate (BER), i.e., the percentage of bits transmitted in error, of data transmitted in 16- or 32-bit chunks with an 8-bit CRC per chunk.



(a) Crosstalk Performance at 16 Bits.

(b) Crosstalk Performance at 32 Bits.

Figure 9: Crosstalk Evaluation. Transmission Bandwidth (blue) and BER (red) for Data Transmitted in 16- and 32-Bit Chunks. 8-Bit CRC Included with Each Chunk.

Figure 9 shows the BER and communication bandwidth (bps) as a function of the binary counter sample period (each sample represents one bit transfer). The x-axis represents the sample period (T), i.e., the amount of time each bit stays on the transmitter. The bandwidth results account for the overhead of re-transmitting data when the received code word is invalid. The CRC bits are not included as part of the bandwidth but are included as part of the BER.

The data demonstrate that both the communication bandwidth and BER are inversely related to the sample period. The relationship between both metrics is very similar and experiences drastic changes if the frequency of transmission is too fast making the system infeasible for data transfer. The most consistent results are achieved when transmitting 16-bits with a sample period between 2.1 ms and 21 ms. As the data width per transmission increases from 16 to 32 bits, the error rate increases significantly. This increase is attributed to the error detecting code being employed. Since the code word is applied to a bit field, all values of that field must be correct, otherwise, re-transmission is necessary. The transmission rate and other parameters can be tuned for target FPGA device.

Improvements The CRC codeword could be directly transmitted between the encoder and decoder instead of using transmission via crosstalk to improve transmission stability and increase performance. This approach circumvents the possibility of errors in the CRC, allowing an increased error tolerance on the data payload. Furthermore, reducing the CRC transmission overhead also increases the raw bandwidth significantly. Our experimentation shows that removing the CRC codeword from the crosstalk transmission stream increases bandwidth by 4× while retaining a BER of nearly zero during 4 KB of data transfer. The improvement allows for a sample period decrease to 1 ms with the same data integrity as the 2.1 ms baseline for 16-bit transmission data width. However, the direct CRC transmission approach does create a physical connection between the transmitter and receiver module that could help a reverse engineer identify and isolate the transmission circuitry.

Environmental and Aging Effects To detect analogous crosstalk communication effects, we analyze the relative differences in wire delays. Note that these delays are induced by the logic value of neighboring wires. For example, the delay decreases in case the neighboring wire is a logic '1'. In addition, wire delay differences are generally unaffected by transistor

aging or environmental effects. More precisely, both sender and receiver transistors are equally affected by the physical phenomena and the detection only focuses on wire delay differences. For more details see Sapatnekar *et al.* [Sap13] and Wang *et al.* [WKK+14].

5.4 BIMAD - Bitstream Manipulation Detection

The deterrence of static and dynamic analysis via hardware obfuscation is only one part of a comprehensive (hardware) obfuscation scheme. Based on our adversary model in Section 4.1, we also assume that a design may be manipulated (e.g., the addition of in-circuit debug circuitry to disclose sensitive information), cf. Figure 10.

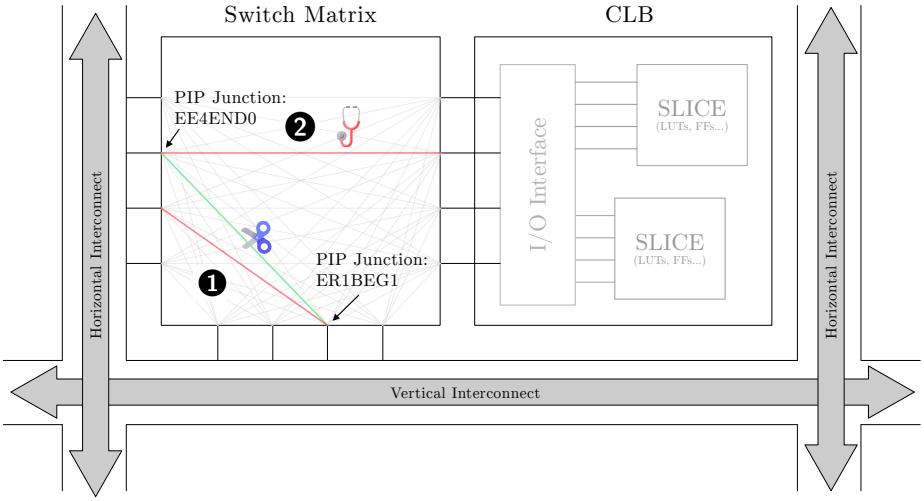


Figure 10: Bitstream Manipulations Targeting Route Through PIP from Source Junction `EE4END0` to Destination Junction `ER1BEG1`.

Before we detail our approach for run-time bitstream integrity checking, we provide fundamental details of the Xilinx 7-Series FPGA architecture and its bitstream format. This information helps illustrate the mechanics of bitstream-level manipulations and our defense strategy.

Bitstream-Level Design Manipulation Modern FPGAs leverage programmable switch matrices to flexibly connect programmable logic/memory resources organized within a grid of tiles. For details on switch matrices, see Section 3.1. The configuration information for logic/memory resources and switch matrices is stored in the bitstream, cf. [Xil18] for Xilinx 7-Series bitstream details. From a high-level perspective, a bitstream consists of *configuration packets* that transmit commands to the FPGA to set up and control the device's initialization process. The configuration of the FPGA fabric, i.e., its programmable logic and routing, resides within the payload of these packets. The fabric's configuration information is split into frames that consist of 101 32-bit words each. In principle, an attacker can add, change, and remove any logic, memory, or programmable interconnect from the configuration, as demonstrated in Section 3.1. Figure 10 presents two potential attack vectors targeting the FPGA's configurable routing. For example, a switch matrix input may be redirected to other source logic or registers ❶ and a switch matrix output can be wiretapped to snoop the data-path at run-time ❷.

Security Consideration We apply self-integrity measures from the software obfuscation domain to hardware to examine a bitstream for manipulations at run-time. In particular, our integrity check consist of two parts: (1) run-time bitstream readback of *selected* frames (this section), and (2) verification of the readback data's validity (Section 7.1). In general, readback and integrity verification of an entire bitstream is discouraged. A bitstream for even medium-sized FPGA families contains multiple megabytes (rendering the readback time impractical).

5.4.1 Design

To better describe bitstream self-integrity checks to support hardware obfuscation, we now present (1) the choice of relevant parts to check for integrity, and (2) a controller that reads back selected parts of the configured bitstream at run-time. This obfuscation primitive requires an understanding of (parts) of the proprietary bitstream file format. Even though our design and implementation focus on Xilinx 7-Series FPGAs, the defense principles can be generalized to other FPGA families and vendors as well.

Choosing Relevant Parts of the Design for Integrity Checks To select which FPGA bitstream configuration frames must be read back, it is necessary to first select the application-specific hardware design portions that are targeted for protection (e.g., a wire path between a source and a destination register). The EDA tool report is then analyzed for detailed place-and-route information to identify the affected hardware components. Afterwards, the documented bitstream file format is leveraged to automatically map this information to configuration bits in the bitstream, i.e., determine which bits have to be set or unset to activate or disable the desired functionality.

Readback Controller To query the regarding configuration bits at run-time, we leverage a readback controller that communicates with the internal FPGA configuration unit, cf. Section 5.1. Configuration bit information is processed by software (Section 7.1).

```
1 $> report_route_status -of_objects [get_nets icap_data_in[0]]
2 CLBLM_L_X20Y23/CLBLM_L.CLBLM_M_AQ->CLBLM_LOGIC_OUTS4
3 INT_L_X20Y23/INT_L.LOGIC_OUTS_L4->>EE4BEG0
4 INT_L_X24Y23/INT_L.EE4END0->>ER1BEG1
5 ...
6 LIOI3_X0Y1/LIOI3.IOI_OLOGIC0_D1->>LIOI_OLOGIC0_OQ
7 LIOI3_X0Y1/LIOI3.LIOI_OLOGIC0_OQ->>LIOI_O0
```

Listing 1: Exemplary Output of Xilinx Vivado's Place-and-Route Report Analysis Tool for Net `icap_data_in[0]`.

External Readout Protection Xilinx offers a feature that allows for the readout of the entire configured bitstream during run-time via JTAG. In the design process, the designer has the option to set the security level so that readout can be prevented. Readout protection is controlled by a single bit in the bitstream, hence manipulation is straightforward. Additionally, the readout status can be determined via the on-chip reconfiguration interface (ICAP). This information can be incorporated into the integrity check if the designer wants to ensure that readout is disabled. Note that the readout protection status is queried in combination with partial reconfiguration. The partial bitstream is only configured if the readout bit is not set to hinder leakage of any design information from the partial design. The readout protection cannot be enabled externally once the bitstream is loaded.

5.4.2 Implementation

Our implementation targets the Xilinx Zynq-7000 XC7Z020 System-on-Chip (SoC) using the Xilinx Vivado toolchain.

Choosing Relevant Parts of the Design for Integrity Checks The bitstream format database provided by *Project X-Ray* [Syme] is leveraged to translate the selected gates and routes to their corresponding configuration bit-patterns in the bitstream. As noted earlier in this subsection, FPGA fabric configuration information in the bitstream is organised in frames of 101 32-bit words each. The database maps features, *e.g.*, a PIP connection or a LUT configuration and their locations within the FPGA's tile grid, onto specific bits within the bitstream. The *Project X-Ray* database utilizes the naming and placement conventions prescribed by Xilinx, thus allowing for the direct correlation of data gathered from Xilinx Vivado to information stored within the database. Having extracted the desired feature names and locations from the EDA tool report, cf. Listing 1, the database enables the generation and storage of (1) *frame address*, (2) *word index within the frame*, and (3) *bit index within the word* tuples for each of the features. This information reveals the bits configuring the respective features and their target state.

Readback Controller Similar to Section 5.1, the ICAPE2 controller (*cf.* Section 5.1.2) is leveraged to handle dynamic reads of the FPGA bitstream configuration. A fixed command sequence that handles mode switching and byte-ordering is used. The required commands, which follow the *SelectMAP* protocol, can be found in Table 6-2 of the Xilinx User Guide [Xil18]. The procedure involves the initialization of the start address and the number of frames to be read via the `FDRO` register.

External Readout Protection The ICAP controller can be queried to obtain the configured status of the readout protection. The readout configuration is stored in the `CTL0` register, which can be read using Xilinx's *SelectMAP* protocol [Xil18].

6 Software Obfuscation Methods

We now provide an overview of the selected software obfuscation methods relevant for this work. Over the years, numerous software obfuscation methods have been proposed, cf. Collberg [CTL97] for a comprehensive overview, and we thus focus on the most prominent techniques (that are also extensively used in commercial obfuscators such as VMProtect [VMP] or Tigress [Tig]) based on virtualization-based obfuscation (Section 6.1), Mixed Boolean Arithmetic (MBA) (Section 6.2), and software integrity measures (Section 6.3).

6.1 Virtual Machine-based Obfuscation

Virtual Machine (VM)-based obfuscation protects software by translating it into a custom Instruction Set Architecture (ISA). The virtual instruction stream is referred to as *bytecode*. The bytecode is then interpreted by a virtual software-based Central Processing Unit (CPU), cf. Figure 11. This obfuscation technique forces a reverse engineer to first understand the ISA as well as VM implementation and then reconstruct the underlying code from the bytecode which is a time-consuming task in practice. In general, virtualization is implemented using a decode-and-dispatch loop that fetches each virtual instruction from memory, decodes it via a large switch-case statement, and then transfers execution to the respective VM handler (that typically performs the operation in native code).

More advanced implementation strategies aim to hide this characteristic software pattern via diverse dispatcher strategies such as indirect dispatch or call dispatch, cf. [Tig]. For virtual instruction sets, an obfuscation designer is free to deviate from classical RISC

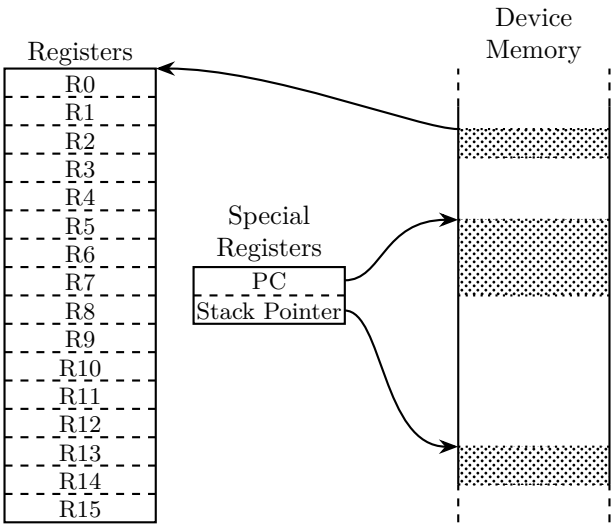


Figure 11: Visualization of a Generic VM Structure.

or CISC architectures. For example, Anckaert *et al.* [AJV06] and Fang *et al.* [FWWH11] proposed designs in which VM instructions may execute whole basic blocks or even complete algorithms. Based on Instruction Set Randomization (ISR), it is moreover possible to change the ISA on-the-fly during execution [CLGJ19]. Even though the effort for static analysis significantly increases with VM-based obfuscation, execution traces can reveal the bytecode location and may lead to further understanding of the virtualization handlers [SLGL09].

6.2 Mixed Boolean Arithmetic

MBA connects arithmetic operations (e.g, addition, multiplication, ...) with logical operations (e.g., exclusive or, bit-shifts, ...) [ZMGJ07]. For example, the computation of $x + y$ is semantically equivalent to $(x \oplus y) + 2 \cdot (x \wedge y)$. MBAs are well-established building blocks for software obfuscation as they enable expression transformation to prevent simplification back to their original form (e.g., to apply instruction substitution or assemble opaque predicates [JRWM15]). MBA expression reduction to simplified expressions is considered NP-hard [ZMGJ07]. With the aforementioned VM-based obfuscation in mind, MBAs are valuable in obfuscating VM handler semantics and thus challenge the reverse engineering of a virtual instruction set architecture even further.

6.3 Software Integrity

Software obfuscation that deters static and dynamic analyses is only one part of a comprehensive protection plan. A reverse-engineer may also learn about the inner workings of the software by manipulation (e.g., changing the control flow). We now describe various strategies to achieve software integrity (in a non-cryptographic sense). In general, self-checksumming can be used to validate code changes within a software program. Horne *et al.* [HMST01] separate self-checksumming into two categories: static checksumming that only runs during start-up and therefore is only invoked once, and dynamic checksumming that runs multiple times during program execution. From a high-level point of view, the responsible code for self-checksumming, a *guard*, is added to the software that examines various properties and thus protects parts of the software. To stop a reverse-engineer from disabling certain guards, each guard may protect other guards to form a circular network of protection, see Figure 12. For example, a guard may read back

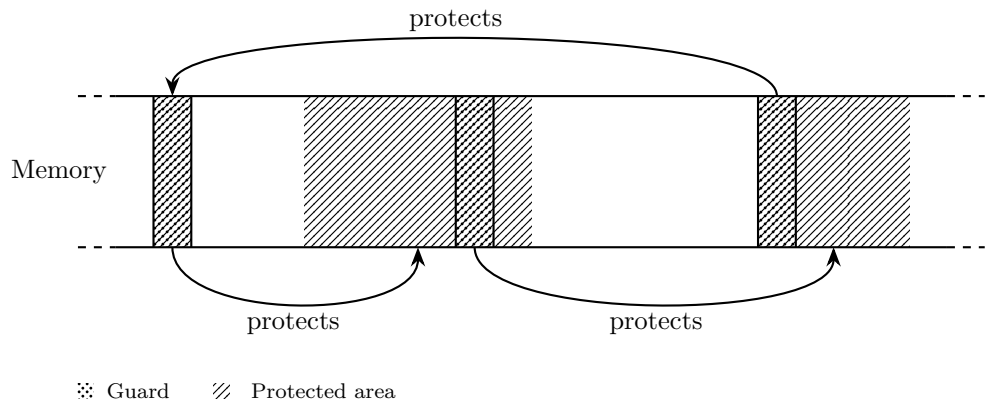


Figure 12: Dynamic Self-Checksumming Realized by a Circular Guard Network Protecting Code and Guards.

parts of code memory or data memory, or even measure execution timings of certain parts. Once manipulation is detected by violation of a property, a guard may invoke a tamper response to counteract the identified manipulation. Note that an immediate response (e.g., system reset) is typically undesirable in practice as the attacker can directly observe what causes the tamper response. Instead, checksum values are often used at a later point in time to attempt to hide immediate effects [TCJ06].

7 LifeLine - Design, Implementation, Case Studies

We now present the design (Section 7.1) and implementation (Section 7.2) of LifeLine, its application in use-cases (Section 7.5), and a security analysis (Section 7.4).

7.1 LifeLine - Design

As introduced in Section 4.2, our key contribution is a system with dynamic control-flow in hardware and dynamic data-flow in software including a tight integration of both worlds. Similar to a standard bitstream encryption scheme flow, our obfuscated authenticator software takes control upon the start of configuration and validates its environment using our hardware obfuscation primitives. If validation succeeds, the encrypted partial bitstream of the protected IP core is decrypted and configured on the FPGA. If the software detects any anomalies, our validation fails quietly and decrypts a false partial bitstream.

Hardware Our hardware setup is comprised of the obfuscation primitives introduced in Section 5. LifeLine uses a soft-core CPU as a main controller, see Figure 13. Note that we used a RISC-V CPU, however, our design is agnostic to the selected ISA. Our reconfiguration controller realizes both the dynamic partial reconfiguration as well as the dynamic bitstream readback for the hardware self-integrity check as detailed in Section 5.1 and Section 5.4. Note that `BIMAD` ensures that security-relevant gates and routes (e.g., from the soft-core to the reconfiguration controller) have been not tampered with. Crosstalk-based transmission of FPGA-unique device IDs safeguards the hardware system from simulation (even for FPGAs that are capable to simulate partial reconfiguration). Note that IDs are used optionally in our system to bind decryption to a specific FPGA device instance. To connect the hardware and software worlds, hardware components are accessible via memory-mapped IO within the soft-core. The encrypted bitstream can either be stored in BRAM or on external non-volatile storage.

Software Our software comprises of the obfuscation primitives introduced in Section 6. Since any soft-core in our design is typically resource-limited, we designed our virtual machine ISA with a low memory footprint in mind. Hence, we opted for an accumulator machine with a 1-byte instruction format. Nonetheless, we equipped handlers of our virtual machine with MBAs to hinder analysis. Therefore, we used an MBA synthesis approach that allows for the generation of MBAs of arbitrary complexity [BCAH17]. To safeguard software integrity, we read back the memory of code and data in software.

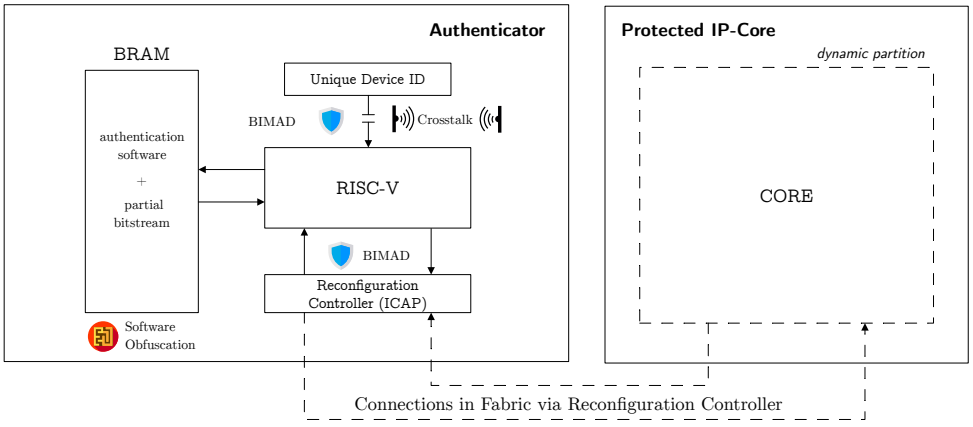


Figure 13: LifeLine Design Overview.

Hardware/Software Co-Obfuscation Since the partial bitstream is encrypted based on a (symmetric) encryption algorithm, we have to generate the cryptographic key in a secure manner. Note that from a cryptographer's perspective key storage without dedicated secure elements can be typically easily circumvented, but with a tight coupling of hardware and software obfuscation we can achieve an efficient security level of a software-based secure key memory in practice. To generate the key, we utilize a hash-based approach where specific device data is accumulated via a hash function, i.e. $k = H(data_1||H(data_2||H(...))$. Note that $data_i$ is hardware or software data source (e.g., crosstalk-based transmitted FPGA ID or code memory location bytes for self-integrity). Hence, manipulation of a single bit of a data source results in a deviation of the cryptographic key (based on the avalanche property of the cryptographic hash function). So malicious changes to the unique hardware ID or the validated bitstreams configuration bits from `BIMAD` have direct consequences on the hash computation and inevitably change the key. Note that our main requirement is that the hardware obfuscation primitives feed data into the computation of k, so that the software black-box cannot be analyzed as a single component and thus creates a strong dependency between hardware and software. To overcome these obstacles an adversary has to simultaneously attack multiple levels from software and hardware to change both the places where computation takes place and the self-integrity checks report their status. Without knowledge when which data feeds into the stream and an easy way to manipulate the software and hardware integrity checks will require considerable effort for which there is currently no known way on how this may be automated.

7.2 LifeLine - Proof-of-Concept Implementation

We now detail our proof-of-concept implementation of LifeLine. Note that we acknowledge run-time limitations as we focused on an implementation with a minimal hardware footprint, so we left out optimizations based on custom instruction set extension or software/hardware co-design accelerators.

Hardware We opted for a custom RISC-V soft core (running at 100 MHz) that uses shared Random Access Memory (RAM)/Read-Only Memory (ROM) (implemented using BRAM blocks) to store its software and the encrypted partial bitstream of the protected hardware design. For development purposes, we also added communication modules to load and inspect software at run-time. Our hardware primitives from Section 5 are instantiated as described before and accessible to the CPU software via memory-mapped Input/Output (I/O). To protect the hardware design from tampering, we opted to guard the connections from the CPU to its peripherals via `BIMAD`. To generate FPGA unique device IDs, we leverage the Xilinx-specific Device DNA [Xil18]. Note that we applied crosstalk-based communication from the ID register to the software as the identifier is only transmitted once. On average, transmitting the 57-bit ID requires around 10ms using the fastest crosstalk configuration with a sample rate of 2.1 ms. For our reconfiguration controller, we use the Xilinx ICAPE2 port as stated in Section 5.1.

The hardware resource overhead of our components is listed in Table 2 and requires overall around 10% of available resources on our target Xilinx Zynq-7000 FPGA.

Software As noted in Section 7.1, we opted for a VM architecture with minimal memory footprint using an accumulator machine and a compact decode-and-dispatch loop. Note that all computations are exclusively carried out between an operand register and the accumulator. We use a 1-Byte instruction format in which 4-Bit are reserved for the operation selection and 4-Bit for the operand register selection. To protect semantics of VM handlers, we synthesized MBAs based on the approach by [SBC+21]. For example, Listing 2 shows the C implementation of the VM handler for a Boolean *AND* operation equipped with an automatically generated MBA. As a proof-of-concept, we leverage a single guard for software integrity that reads back all code and data from software memory.

```
1 switch(opcode) {
2 ...
3 case AND: {
4     // r0 is the accumulator register and ry the operand register
5     r0 =  (~ ((((0x2 << ((~ (- ((r0 + 0xffffffff) ^ 0xffffffff))) & 0x1f))
          | (~ ((~ (((~ r0) ^ (~ ((- (((- r0) + 0x2) << ((((- r0) + 0x2) <<
          0x2) & 0x2))) + 0x2))) & (~ (- (~ (ry ^ (~ ry)))))))) & (~ (~ (((~
          ((~ ((~ ((~ ((~ ((((- r0) + 0x1) | (r0 | ((r0 ^ (~ (r0 | (r0 & (- (
          ry + 0xffffffff)))))) | 0x2))) & ((r0 ^ (~ (r0 | (r0 & (- (ry + 0
          xffffffff)))))) | 0x2))) | (((r0 ^ (~ (r0 | (r0 & (- (ry + 0
          xffffffff)))))) | 0x2) << ((- ((r0 ^ (~ (r0 | (r0 & (- ((- ((- ((~
          ((~ (r0 | ((~ (ry & 0x2)) | 0x2))) | ry)) & ry)) + 0x1)) + ry))))))
           | 0x2)) & 0x1f)))) | ((r0 ^ (~ (r0 | (r0 & (- (ry + 0xffffffff))))
          )) | 0x2))) | (~ ry))) | (~ ry))) ^ ry) & ry)))))) & (~ ((~ ((~ r0)
           ^ (~ (- (~ ((- r0) ^ 0xffffffff)))))) & (((r0 ^ (- ((- ((- (~ (-
          r0))) + 0xffffffff)) + 0x1))) | 0x2) & ry)))) | (~ r0)));
6     break;
7 }
8 ...
9 }
```

Listing 2: Example: A hardened VM handler for $r0 = r0 \wedge ry$ equipped with an MBA expression.

Hardware/Software Co-Obfuscation We implemented the hash-based generation of a decryption key using Simon 32/64 (in Davies-Meyer mode) where the cryptographic round-function is implemented using our hardened VM. Note that we compiled our C-implementation of the virtual machine and hash-based key generation, and decryption using an adapted `riscv-gcc` toolchain and optimized the software for performance with `O3`. We now provide an overview of the run-time of different hardware/software components including interaction based on the computation of the data hash with the protected Simon

implementation. Our crosstalk-based readout of the FPGA ID requires around 10ms on average as the analog crosstalk behavior features an irregular run-time due to required error detection. Note that the transmission can be enabled upon FPGA boot-up and the value read later. For the self-integrity readback and software check via `BIMAD`, we require around 3 ms on average for a route for both eavesdropping and manipulation detection. In our experiments, we selected a route spanning across 4 PIPs, which would have to be checked for manipulation. To safeguard for eavesdropping 52 additional configurations of the selected PIP have to be added, to ensure no additional destinations can be added. The amount of additional PIPs to check, depends however on the regarding PIP. Our tool is able to identify all necessary PIPs for the eavesdropping connection. Each configuration consists of around 1-5 bits in the bitstream that have to be checked for that matter.

The actual encryption/decryption uses a hash-based approach. The key is derived from multiple hardware and software values (cf. Section 7.1). It is used in combination with the Simon cipher in CTR mode. The output is then used as a seed for ISAAC, a fast software Pseudo Random Number Generator (PRNG) [Jen96], to generate a 1 kB keystream. The partial bitstream is split into 1 kB chunks and each chunk is decrypted using a newly generated keystream.

7.3 Area and Performance Overhead

Our proof-of-concept induce minor hardware overhead, as seen in Table 2, because we only require a simplistic soft-core with two additional peripherals. Most security mechanisms are implemented in software and requires 135 KB of memory space, i.e. BRAM, while 62 KB of the 135 KB are used to store the partial bitstream.

Table 2: Hardware Resource Overhead for Xilinx Zynq-7000 FPGA (ZedBoard).

	LUT	**FF**	**BRAM**	**LUT-RAM**
RISC-V	2643	2743	32	160
Crosstalk Communication Primitive	330	348	0	0
ICAP Controller	146	118	0	0
Total Used	3119	3209	32	160
Total Available	53 200	106 400	140	17 400

LifeLine creates a *one-time* start-up penalty of 560 ms for the key generation, *cf.* Table 3, and 591 ms for the decryption. After the authentication process LifeLine does not influence the user design except for the static area overhead. The majority of the time is spent on checking the environment integrity and to perform the partial bitstream decryption. Note that the decryption itself is hardened with software obfuscation and thus is by nature slower than a native implementation.

Table 3: Time overhead for key generation of LifeLine at 100 MHz

	Average Execution Time
DNA Transmission via Crosstalk	10 ms
57 `BIMAD` Checks	228 ms
Software Self-Integrity Check	260 ms
Key Accumulation via Hashing	34 ms
Partial Reconfiguration	28 ms
Total	**560 ms**

We want to emphasize that our soft-core does not implement any optimizations such as a pipeline and only runs at 100 MHz. Hence, the use of a different, i.e. optimized soft-core may drastically decrease the start-up penalty (while increasing the hardware area overhead).

7.4 Security Analysis

In the following we detail how any missing obfuscation primitive would yield a circumvention of the protection and thus it shows that our selected primitives for hardware and software obfuscation are minimal but offer adequate protection. Note that in general our coupling of hardware and software via hash-based key-generation does not allow for single-component analysis as described in the following.

Hardware On the hardware side, as an attacker, we deter any static analysis by partial reconfiguration as the design is only unfold at run-time. In this way, an attacker is resort to simulation or on-chip debugging (coupled with malicious design alterations) to analyze the system. To this end, an attacker may attempt to alter the routing inside the FPGA to eavesdrop on the values exchanged between the softcore peripherals or to reroute the ICAP peripheral to his own malicious implementation. To counteract such manipulation, we read back selected parts of the bitstream information and incorporate these values in the key derivation (cf Section 5.4). We want to emphasize that an adversary may attempt to alter the software (e.g., to overwrite returned values), however, since our software is again protected with obfuscation measures and feedback to the hardware (via covert communication) reverse-engineering and manipulation becomes a *chicken-and-egg* problem for an attacker. Note that in principle an attacker can analyze hardware and software simultaneously, but such a software-hardware co-deobfuscation cannot be automated to the best of our knowledge (with respect to state-of-the-art tools). Moreover, our hardware implementation can also feature CPU implementations with randomized instruction set architectures [FRB+18] so that a reverse-engineer would have to also reverse-engineer the instruction set architecture first (required to disassemble the obfuscated software). In addition, an attacker may attempt to insert a malicious reconfiguration controller into the system that correctly answers all read requests (even though the configured routes differ from the returned values). To this end, a malicious reconfiguration controller has to store the original bitstream (which requires significant memory space), and it must meet the same timing requirements as its original counterpart. Note that our implementation requires precise timing and thus acts similar to a run-time integrity check via the entire *SelectMAP* protocol. In summary, we want to emphasize that our selected arsenal of hardware obfuscation primitives is minimal, i.e. when one of the primitives is removed an attacker may find a semi-automated approach to attack the system.

Software On the software side, VM-based obfuscation requires an attacker to the tedious task and manual to understand the VM architecture. Even though semi-automated approaches to reverse-engineer VM-based obfuscation exists (e.g., symbolic execution or program synthesis [BCAH17]), the effectiveness of the attacks typically decrease with an increasing complex handler. Hence, without the additional drastically-increasing syntactic complexity by MBAs our simplistic VM architecture would be targeted by semi-automated approaches. We want to note that with automatically generated MBAs and a flexible VM architecture we are able to generate a diverse set of implementations. Note that without the software integrity measures by guard networks an adversary may be able to introduce breakpoints or replace cryptographic operations with simplified linear expressions to observe divergent behavior and draw conclusions about utilized decryption keys. However, patching cyclic guard networks is a non-trivial task in practice, cf. Section 6.3. Moreover, we want to emphasize that we are free to choose the selected decryption algorithm and use secure but proprietary randomized variants (e.g., an Advanced Encryption Standard (AES) with a randomized Sbox or a proprietary key schedule still offers the same security guarantees). In this way, the proprietary nature of the cipher implementation renders side-channel attacks meaningless as the adversary cannot generate correct key hypotheses. In summary, we want to emphasize that our selected arsenal of software obfuscation primitives is minimal, i.e. when one of the primitives is removed an attacker may find a semi-automated approach to attack the system.

7.4.1 Towards a Systematic Security Analysis of the Proposed Scheme

A systematic security analysis of obfuscation metric is hard to perform in a general sense, because of the *human factor* (particularly different skill levels and approaches) introduced by the reverse engineer that cannot be quantified so far [BWA+20, WAH+19, HP18]. In 1997 Collberg *et al.* introduced certain criteria for software transformation/obfuscation which can be partly applied to hardware in order to evaluate the strength of the obfuscation method. Three main criteria as *measurement of quality* are characterized: (1) the amount of obscurity added to the program (potency), (2) the difficulty to break for an automatic deobfuscator (resilience), and (3) the computational overhead added by the obfuscation (cost) [CTL98, HP18]. Note that the cost criteria is discussed in Section 7.3.

Potency Our deep coupling of various software and hardware obfuscation techniques forces the attacker to carry out the attack simultaneously at both hardware and software levels. He cannot attack the software without analyzing/attacking the hardware nor attack the hardware without control and a deep understanding of the software. Thus, an interdisciplinary team of experts in both hardware reverse engineering and software reverse engineering is required. To successfully attack the system, the self-integrity checks by `BIMAD` on both hardware and software have to be circumvented first. While the checks are issued from the software and then performed in hardware, its hardware results are again handled in software and influence the key derivation used for decryption. In particular, the attacker does not know what checks are performed at which point in time. As mentioned before, an attacker may simulate the response from the hardware side, however, this requires a malicious ICAP controller while meeting timing requirements mandated by the protocol. In addition, the device ID transmission (obfuscated with the help of crosstalk) cannot be simulated and has to be carried out on the device. Again the device ID is also used in software as part of the key derivation and thus adds another coupling of both software and hardware. Once the attacker overcomes these barriers, he is faced with reverse-engineering of software that is equipped with several well-researched obfuscation mechanisms, as mentioned in the preceding section.

Resilience LifeLine has high resilience, in regards to state-of-the-art automated deobfuscation approaches. Currently, there are not completely automated methods for hardware reverse engineering, in particular gate-level netlist reverse engineering. Even though various automated methods exist (e.g., to detect FSMs [STGR10, FWD+18] or registers [AHT+20]), these methods only provide support in the manual analysis phase. Even though various automated methods exist to extract information from an (obfuscated) software binary (e.g., symbolic execution or program synthesis [BCAH17]), reverse engineering of virtualization-based obfuscation generally includes several crucial manual steps (before automated methods can be applied) such as examination of the virtual machine and design of its intermediate language [SBC+21].
Based on these criteria LifeLine provides strong additional security. The creation of a strong bond between software and hardware significantly increases the effort of the attacker. He is not able to analyze all components in isolation but has to attack the whole system on different levels at once, making it a time-consuming task.

7.5 Use-Cases of LifeLine

We now outline two different application use-cases for LifeLine to implement (1) a bitstream encryption feature and (2) a pay-per-use IP core license scheme.

Use-Case: Bitstream Encryption One application of LifeLine is to implement bitstream protection that retrofits a security solution to FPGA families with vulnerable bitstream encryption schemes. To achieve bitstream protection, the design that is supposed to be protected has to be put into the *dynamic partition*, so it can be dynamically reconfigured by

LifeLine. To this end, we require to declare a re-configurable region that fills almost all of the FPGA (except for a small static region designated for LifeLine). However, this imposes design restrictions as re-configurable partitions inside of other re-configurable partitions are not properly supported yet (as it requires custom tooling such as GoAhead [BKT12]).

Thus, LifeLine provides protection against reverse engineering (and targeted manipulation). With the bitstream encrypted and obfuscated in software, the bitstream will only be applied on the allowed devices with the correct device ID. Our various levels of obfuscation ensure that the adversary has to take a considerable effort and attack LifeLine at several points in hardware and software simultaneously. A read-out of the configured bitstream at run-time, i.e. after LifeLine performed the bitstream decryption, is also not possible, even if the original bitstream of LifeLine (containing the design that is flashed on start-up of the design) is modified by toggling the read-out protection bit since LifeLine sets the read-out protection bit again internally (*cf.* Section 5.4).

Use-Case: IP Core Licensing In addition to encryption of a whole hardware design, LifeLine can be used to build an IP core license scheme with a binding to specific FPGA instances. As noted in Section 7.1, we leverage the unique FPGA device ID as key-dependent data. Hence the software (or in this case the license file) cannot be transferred from one FPGA to another (with a different device ID) without prior reverse-engineering and manipulation of both hardware and software. Hence, an IP core provider can use issue licenses based on LifeLine equipped with the device ID and thus control in a fine-granular way on how many devices an IP user is allowed to run the protected IP core.

8 Discussion

In the following, we will discuss the generality and implications of our approach and provide a prospect for future research directions.

Generality Even though our techniques and our proof-of-concept focus on Xilinx SRAM-based FPGAs, our technique can be adapted to other SRAM-based FPGAs of other vendors as well. For example, Intel/Altera feature a unique hardware ID and partial reconfiguration, but with the drawback (from a security engineer's point of view) that partial reconfiguration can be simulated [Cor19]. Moreover, `BIMAD` and crosstalk are not restricted to the SRAM programming technology and can be implemented on flash-based FPGAs to deter reverse engineering. As noted before our software obfuscation is not restricted to RISC-V and any ISA can be utilized as a soft-core. Hardware/software co-obfuscation can therefore be a potential solution to the particular problem of FPGA bitstream security on insecure platforms.

Implications In Section 7 we demonstrated the effectiveness of our defenses with respect to static and dynamic analysis for both hardware and software. Hence, we increase the effort of an attack by a large factor, since it requires the attacker to have knowledge in both hardware and software reverse-engineering. Thus, obfuscation in general, especially hardware/software co-obfuscation, can therefore be a potential solution to the pressing problem of real-world FPGA security. It should be noted that the interplay between software and hardware must be carefully designed in order for the scheme to be effective such that it can not easily be circumvented. Moreover, our work demonstrated again that reverse-engineering and targeted manipulation of FPGA bitstreams it not only a theoretic attack vector, cf. Section 3.1, and various security schemes can be circumvented for vulnerable FPGA families.

Limitations We acknowledge the limitation that in case FPGA vendors offer a secure cryptographic implementation to decrypt the bitstream, the hardware bitstream encryption scheme should be used rather than LifeLine. However, for none or insecure bitstream encryption schemes, our solution yields an effective security level for practical applications. From a cryptography point of view we acknowledge the limitation of our insecure-by-design obfuscation based approach, however, obfuscation is the only alternative to retrofit any security to vulnerable FPGAs.

Additional Obfuscation Primitives PUFs are an alternative solution that offers similar security properties such as anti-simulation. Especially on devices that do not feature a unique identifier, such as Xilinx's device DNA, a PUF could be used instead of or even in addition to providing additional security features. As PUFs require a higher implementation and integration effort in general, we decided not to include them. Especially for the license-based approach, where an IP-Core is potentially integrated into tens of thousands of devices, the designer has to perform extensive configuration and evaluation during the enrollment phase for each device. Note that various strong PUFs (in a cryptographic sense) have been shown to be prone to modeling attacks [VPPK16].

Comparison to Academic Solutions Partial reconfiguration run-time environments have been proposed in other security solutions before. For example, Huss *et al.* [HS17] use a hardware/software co-design to create a software-managed mutating environment with the goal to minimize power side-channels, i.e. by mapping implementation parts to different processing units with *slightly* different power characteristics at run-time. Maes *et al.* [MSV12] leverage partial reconfiguration to create a pay-per-use licensing scheme. A similar scheme as been proposed by Zhang *et al.* [ZC14]. Compared to LifeLine, their startup penalty and resource usage are neglectable, since they only require an (encrypted) decryption core with an embedded key. However, analogously to many other schemes, this solution requires the usage of a TTP and an enrollment phase. This is not necessary for LifeLine, because it does not use the built-in bitstream decryption engine (that is vulnerable to automated implementation attacks for certain families) and thus does not require an enrollment phase. In addition, the IP vendor can host all necessary services. Guajardo *et al.* [GKST07] use a PUFs instead of the device-specific key and built-in decryption engine. Therefore, they have also feature a neglectable startup penalty at the cost of a TTP protocol. Kumar *et al.* [KSM+17] employ a more flexible scheme similar to LifeLine without the requirement of a TTP. However, the scheme is completely implemented in hardware compared to the hardware/software co-obfuscation by LifeLine. Note that Kumar *et al.* assume that a hash and a decryption engine are located on-chip and since the IP core is encrypted using *RSA* a significant time overhead is added. Furthermore, some of the above schemes can be broken, *cf.* Section 3.2.

Future Work In this work, we demonstrated crosstalk and partial reconfiguration as an anti-simulation method that also helps to deter static analysis. In future work, it should be explored what other hardware primitives may be suitable as anti-simulation methods, especially if there is improved tooling available to simulate partial reconfiguration. Moreover, another direction for future research is the combination of ISA and hardware obfuscation features in order to challenge software reverse engineering similar to Fyrbiak *et al.* [FRB+18]. In particular, ISA diversification with dynamic self-modification may be a fruitful research path to harden software obfuscation features.

9 Conclusion

Numerous attacks on the bitstream encryption schemes of SRAM-based FPGAs have demonstrated that FPGA are in need for a solution to provide sound protection even under the assumption that its cryptographic implementation is vulnerable. In this work, we first reviewed various FPGA design protection schemes from academia and industry and demonstrated how to straightforward design manipulation on the bitstream-level enables to circumvent protection in automated manner. We then provided design and implementation of novel hardware obfuscation primitives by leverage of dynamic hardware reconfiguration and configuration readback coupled with bitstream architecture details, and physical effects by means of crosstalk. Based on the foundation of our primitives, we designed and implemented LifeLine a hardware design protection for FPGAs using hardware/software co-obfuscated cryptography. We demonstrated that our defense offers effective protection in a realistic adversary model, requires minimal integration effort, and retrofits to already deployed and so far vulnerable systems.

Acknowledgments

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy (EXC 2092 CASA 390781972), through ERC grant 695022, BMBF project grant VE-HEP (16KIS1345) and NSF grant CNS-1563829.

References

[AHT+20] Nils Albartus, Max Hoffmann, Sebastian Temme, Leonid Azriel, and Christof Paar. DANA universal dataflow analysis for gate-level netlist reverse engineering. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(4):309–336, 2020.

[AJV06] Bertrand Anckaert, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In Moti Yung, Kaoru Kurosawa, and Reihaneh Safavi-Naini, editors, *Proceedings of the Sixth ACM Workshop on Digital Rights Management, Alexandria, VA, USA, October 30, 2006*, pages 47–58. ACM, 2006.

[BBS19] Michaela Brunner, Johanna Baehr, and Georg Sigl. Improving on state register identification in sequential hardware reverse engineering. In *IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2019, McLean, VA, USA, May 5-10, 2019*, pages 151–160. IEEE, 2019.

[BCAH17] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 643–659. USENIX Association, 2017.

[BKT12] Christian Beckhoff, Dirk Koch, and Jim Torresen. Go ahead: A partial reconfiguration framework. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 37–44. IEEE, 2012.

[BSH12] Florian Benz, André Seffrin, and Sorin A. Huss. Bil: A tool-chain for bitstream reverse-engineering. In Dirk Koch, Satnam Singh, and Jim Tørresen, editors, *22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, August 29-31, 2012*, pages 735–738. IEEE, 2012.

[BWA+20] Steffen Becker, Carina Wiesen, Nils Albartus, Nikol Rummel, and Christof Paar. An exploratory study of hardware reverse engineering - technical and cognitive processes. In Heather Richter Lipford and Sonia Chiasson, editors, *Sixteenth Symposium on Usable Privacy and Security, SOUPS 2020, August 7-11, 2020*, pages 285–300. USENIX Association, 2020.

[CLGJ19] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. Dynopvm: Vm-based software obfuscation with dynamic opcode mapping. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *Applied Cryptography and Network Security - 17th International Conference, ACNS 2019, Bogota, Colombia, June 5-7, 2019, Proceedings*, volume 11464 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2019.

[Cor19] Altera Corporation. Intel partial reconfiguration ip core. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-partrecon.pdf, Apr 2019. Accessed: 2021-04-09.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CTL98] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In David B. MacQueen and Luca Cardelli, editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 184–196. ACM, 1998.

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

[EHP19] Susanne Engels, Max Hoffmann, and Christof Paar. The end of logic locking? A critical view on the security of logic locking. *IACR Cryptol. ePrint Arch.*, 2019:796, 2019.

[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.

[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.

[FRB+18] Marc Fyrbiak, Simon Rokicki, Nicolai Bissantz, Russell Tessier, and Christof Paar. Hybrid obfuscation to protect against disclosure attacks on embedded microprocessors. *IEEE Trans. Computers*, 67(3):307–321, 2018.

[FWD+18] Marc Fyrbiak, Sebastian Wallat, Jonathan Déchelotte, Nils Albartus, Sinan Böcker, Russell Tessier, and Christof Paar. On the difficulty of fsm-based hardware obfuscation. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):293–330, 2018.

[FWR+20] Marc Fyrbiak, Sebastian Wallat, Sascha Reinhard, Nicolai Bissantz, and Christof Paar. Graph similarity and its applications to hardware security. *IEEE Trans. Computers*, 69(4):505–519, 2020.

[FWS+18] Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier, and Christof Paar. HAL-The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[FWWH11] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage binary code obfuscation using improved virtual machine. In Xuejia Lai, Jianying Zhou, and Hui Li, editors, *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings*, volume 7001 of *Lecture Notes in Computer Science*, pages 168–181. Springer, 2011.

[GKST07] Jorge Guajardo, Sandeep S. Kumar, Geert Jan Schrijen, and Pim Tuyls. FPGA intrinsic pufs and their use for IP protection. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer, 2007.

[GRE18] Ilias Giechaskiel, Kasper B. Rasmussen, and Ken Eguro. Leaky wires. *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, May 2018.

[HAL] HAL. HAL — The Hardware Analyzer. [Online]. Available: https://github.com/emsec/hal.

[HAN18] Noor Ahmad Hazari, Faris Alsulami, and Mohammed Niamat. Fpga ip obfuscation using ring oscillator physical unclonable function. In *NAECON 2018-IEEE National Aerospace and Electronics Conference*, pages 105–108. IEEE, 2018.

[HMST01] Bill G. Horne, Lesley R. Matheson, Casey Sheehan, and Robert Endre Tarjan. Dynamic self-checking techniques for improved tamper resistance. In Tomas Sander, editor, *Security and Privacy in Digital Rights Management, ACM CCS-8 Workshop DRM 2001, Philadelphia, PA, USA, November 5, 2001, Revised Papers*, volume 2320 of *Lecture Notes in Computer Science*, pages 141–159. Springer, 2001.

[HP18] Max Hoffmann and Christof Paar. Stealthy opaque predicates in hardware - obfuscating constant expressions at negligible overhead. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):277–297, 2018.

[HS17] Sorin A. Huss and Marc Stöttinger. *A Novel Mutating Runtime Architecture for Embedding Multiple Countermeasures Against Side-Channel Attacks*, pages 165–184. Springer International Publishing, Cham, 2017.

[Jen96] Robert J Jenkins. Isaac. In *International Workshop on Fast Software Encryption*, pages 41–49. Springer, 1996.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[KHPC19] Jatin Kataria, Rick Housley, Joseph Pantoga, and Ang Cui. Defeating cisco trust anchor: A case-study of recent advancements in direct FPGA bitstream manipulation. In Alex Gantman and Clémentine Maurice, editors, *13th USENIX Workshop on Offensive Technologies, WOOT 2019, Santa Clara, CA, USA, August 12-13, 2019*. USENIX Association, 2019.

[KSM+17] K. Sudeendra Kumar, Sauvagya Ranjan Sahoo, Abhishek Mahapatra, Ayas Kanta Swain, and Kamala Kanta Mahapatra. A flexible pay-per-device licensing scheme for FPGA IP cores. In *2017 IEEE Computer Society Annual Symposium on VLSI, ISVLSI 2017, Bochum, Germany, July 3-5, 2017*, pages 677–682. IEEE Computer Society, 2017.

[LG97] Eric Lechner and Steve Guccione. The java environment for reconfigurable computing. In Wayne Luk, Peter Y. K. Cheung, and Manfred Glesner, editors, *Field-Programmable Logic and Applications, 7th International Workshop, FPL '97, London, UK, September 1-3, 1997, Proceedings*, volume 1304 of *Lecture Notes in Computer Science*, pages 284–293. Springer, 1997.

[LGS+13] Wenchao Li, Adrià Gascón, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, pages 67–74. IEEE Computer Society, 2013.

[LPF10] Shaoshan Liu, Richard Neil Pittman, and Alessandro Forin. Minimizing partial reconfiguration overhead with fully streaming DMA engines and intelligent ICAP controller (abstract only). In Peter Y. K. Cheung and John Wawrzynek, editors, *Proceedings of the ACM/SIGDA 18th International Symposium on Field Programmable Gate Arrays, FPGA 2010, Monterey, California, USA, February 21-23, 2010*, page 292. ACM, 2010.

[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.

[McE01] Kenneth S. McElvain. Methods and apparatuses for automatic extraction of finite state machines, 2001.

[MD20] Michail Moraitis and Elena Dubrova. Bitstream modification attack on SNOW 3g. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 1275–1278. IEEE, 2020.

[MJTZ16] Travis Meade, Yier Jin, Mark Tehranipoor, and Shaojie Zhang. Gate-level netlist reverse engineering for hardware security: Control logic register identification. In *IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016*, pages 1334–1337. IEEE, 2016.

[MKP12] Amir Moradi, Markus Kasper, and Christof Paar. Black-box side-channel attacks highlight the importance of countermeasures - an analysis of the xilinx virtex-4 and virtex-5 bitstream encryption mechanism. In Orr Dunkelman, editor, *Topics in Cryptology - CT-RSA 2012 - The Cryptographers' Track at*

the RSA Conference 2012, San Francisco, CA, USA, February 27 - March 2, 2012. Proceedings, volume 7178 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2012.

[MOPS13] Amir Moradi, David F. Oswald, Christof Paar, and Pawel Swierczynski. Side-channel attacks on the bitstream encryption mechanism of altera stratix II: facilitating black-box analysis using software reverse-engineering. In Brad L. Hutchings and Vaughn Betz, editors, *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*, pages 91–100. ACM, 2013.

[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.

[MSV12] Roel Maes, Dries Schellekens, and Ingrid Verbauwhede. A pay-per-use licensing scheme for hardware IP cores in recent sram-based fpgas. *IEEE Trans. Inf. Forensics Secur.*, 7(1):98–108, 2012.

[MZJ16] Travis Meade, Shaojie Zhang, and Yier Jin. Netlist reverse engineering for high-level functionality reconstruction. In *21st Asia and South Pacific Design Automation Conference, ASP-DAC 2016, Macao, Macao, January 25-28, 2016*, pages 655–660. IEEE, 2016.

[Ngu16] Jean-Francois Nguyen. Analysing the Bitstream of Altera's MAX-V CPLDs. https://lse.epita.fr/lse-summer-week-2016/slides/lse-summer-week-2016-07-maxv_cpld.pdf, July 2016.

[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.

[PHK17] Khoa Dang Pham, Edson L. Horta, and Dirk Koch. BITMAN: A tool and API for FPGA bitstream manipulations. In David Atienza and Giorgio Di Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 894–897. IEEE, 2017.

[RPD+18] Chethan Ramesh, Shivukumar Basanagouda Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel Holcomb, and Russell Tessier. Fpga side channel attacks without physical access. pages 45–52, 04 2018.

[Sap13] Sachin S. Sapatnekar. What happens when circuits grow old: Aging issues in CMOS design. In *2013 International Symposium on VLSI Design, Automation, and Test, VLSI-DAT 2013, Hsinchu, Taiwan, April 22-24, 2013*, pages 1–2. IEEE, 2013.

[SBC+21] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. Loki: Hardening code obfuscation against automated attacks. *arXiv preprint arXiv:2106.08913*, 2021.

[sec] secworks. SHA-256 verilog Core. [Online]. Available: https://github.com/secworks/sha256.

[SFKP15] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, and Christof Paar. FPGA trojans through detecting and weakening of cryptographic primitives. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(8):1236–1249, 2015.

[SLGL09] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 94–109. IEEE Computer Society, 2009.

[SMOP15] Pawel Swierczynski, Amir Moradi, David F. Oswald, and Christof Paar. Physical security evaluation of the bitstream encryption mechanism of altera stratix II and stratix III fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 7(4):34:1–34:23, 2015.

[STGR10] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. A highly efficient method for extracting fsms from flattened gate-level netlist. In *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*, pages 2610–2613. IEEE, 2010.

[STP+13] Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. Reverse engineering digital circuits using functional analysis. In Enrico Macii, editor, *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1277–1280. EDA Consortium San Jose, CA, USA / ACM DL, 2013.

[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.

[Syma] Project SymbiFlow. fasm2bels. [Online]. Available: https://github.com/emsec/hal.

[Symb] Project SymbiFlow. Proejct IceStorm. [Online]. Available: https://github.com/emsec/hal.

[Symc] Project SymbiFlow. Project Trellis. [Online]. Available: https://github.com/SymbiFlow/prjtrellis.

[Symd] Project SymbiFlow. Project U-Ray. [Online]. Available: https://github.com/SymbiFlow/prjuray.

[Syme] Project SymbiFlow. Project X-Ray. [Online]. Available: https://github.com/SymbiFlow/prjxray.

[Sym18] SymbiFlow. Project X-Ray, 2018.

[TCJ06] Gang Tan, Yuqun Chen, and Mariusz H. Jakubowski. Delayed and controlled failures in tamper-resistant software. In Jan Camenisch, Christian S. Collberg, Neil F. Johnson, and Phil Sallee, editors, *Information Hiding, 8th International Workshop, IH 2006, Alexandria, VA, USA, July 10-12, 2006. Revised Selcted Papers*, volume 4437 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2006.

[Tig] Tigress. Tigress. [Online]. Available: https://tigress.wtf/.

[USZ+20] Florian Unterstein, Tolga Sel, Thomas Zeschg, Nisha Jacob, Michael Tempelmeier, Michael Pehl, and Fabrizio De Santis. Secure update of fpga-based secure elements using partial reconfiguration. *IACR Cryptol. ePrint Arch.*, 2020:833, 2020.

[Ver] Veripool. Verilator, the fastest Verilog/SystemVerilog simulator. [Online]. Available: https://www.veripool.org/wiki/verilator.

[VF18] Kizheppatt Vipin and Suhaib A. Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4):72:1–72:39, 2018.

[VMK+15] Jo Vliegen, Nele Mentens, Dirk Koch, Dries Schellekens, and Ingrid Verbauwhede. Practical feasibility evaluation and improvement of a pay-per-use licensing scheme for hardware IP cores in xilinx fpgas. *J. Cryptogr. Eng.*, 5(2):113–122, 2015.

[VMP] VMProtect. VMProtect. [Online]. Available: https://vmpsoft.com.

[VPPK16] Arunkumar Vijayakumar, Vinay C. Patil, Charles B. Prado, and Sandip Kundu. Machine learning resistant strong PUF: possible or a pipe dream? In William H. Robinson, Swarup Bhunia, and Ryan Kastner, editors, *2016 IEEE International Symposium on Hardware Oriented Security and Trust, HOST 2016, McLean, VA, USA, May 3-5, 2016*, pages 19–24. IEEE Computer Society, 2016.

[WAH+19] Carina Wiesen, Nils Albartus, Max Hoffmann, Steffen Becker, Sebastian Wallat, Marc Fyrbiak, Nikol Rummel, and Christof Paar. Towards cognitive obfuscation: impeding hardware reverse engineering based on psychological insights. In Toshiyuki Shibuya, editor, *Proceedings of the 24th Asia and South Pacific Design Automation Conference, ASPDAC 2019, Tokyo, Japan, January 21-24, 2019*, pages 104–111. ACM, 2019.

[WFSP17] Sebastian Wallat, Marc Fyrbiak, Moritz Schlögel, and Christof Paar. A look at the dark side of hardware reverse engineering - a case study. In *IEEE 2nd International Verification and Security Workshop, IVSW 2017, Thessaloniki, Greece, July 3-5, 2017*, pages 95–100. IEEE, 2017.

[WH15] Neil HE Weste and David Harris. *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[WKK+14] Xiaofei Wang, John Keane, Tony Tae-Hyoung Kim, Pulkit Jain, Qianying Tang, and Chris H. Kim. Silicon odometers: Compact in situ aging sensors for robust system design. *IEEE Micro*, 34(6):74–85, 2014.

[Xil18] Xilinx. 7 series fpgas configuration: User guide. https://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf, August 2018. Accessed: 2021-04-09.

[Xil21] Xilinx. Vivado design suite user guide: Dynamic function exchange. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_2/ug909-vivado-partial-reconfiguration.pdf, February 2021. Accessed: 2021-04-09.

[ZAT06] Daniel Ziener, Stefan Assmus, and Jürgen Teich. Identifying FPGA ip-cores based on lookup table content analysis. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications (FPL), Madrid, Spain, August 28-30, 2006*, pages 1–6. IEEE, 2006.

[ZC14] Li Zhang and Chip-Hong Chang. A pragmatic per-device licensing scheme for hardware IP cores on sram-based fpgas. *IEEE Trans. Inf. Forensics Secur.*, 9(11):1893–1905, 2014.

[ZLL+13] Jiliang Zhang, Yaping Lin, Yongqiang Lyu, Gang Qu, Ray C. C. Cheung, Wenjie Che, Qiang Zhou, and Jinian Bian. FPGA IP protection by binding finite state machine to physical unclonable function. In *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*, pages 1–4. IEEE, 2013.

[ZMGJ07] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *Information Security Applications, 8th International Workshop, WISA 2007, Jeju Island, Korea, August 27-29, 2007, Revised Selected Papers*, volume 4867 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2007.