On the Difficulty of FSM-based Hardware Obfuscation

Marc Fyrbiak[1], Sebastian Wallat[2], Jonathan Déchelotte[3], Nils Albartus[1], Sinan Böcker[1], Russell Tessier[2] and Christof Paar[1,2]

[1] Horst Görtz Institute for IT-Security, Ruhr-Universität Bochum, Germany
[2] University of Massachusetts Amherst, MA, USA
[3] University of Bordeaux, France

{marc.fyrbiak,nils.albartus,sinan.boecker,christof.paar}@rub.de
{swallat,tessier}@umass.edu,jonathan.dechelotte@ims-bordeaux.fr

Abstract. In today's Integrated Circuit (IC) production chains, a designer's valuable Intellectual Property (IP) is transparent to diverse stakeholders and thus inevitably prone to piracy. To protect against this threat, numerous defenses based on the obfuscation of a circuit's control path, i.e. Finite State Machine (FSM), have been proposed and are commonly believed to be secure. However, the security of these sequential obfuscation schemes is doubtful since realistic capabilities of reverse engineering and subsequent manipulation are commonly neglected in the security analysis. The contribution of our work is threefold: First, we demonstrate how high-level control path information can be automatically extracted from third-party, gate-level netlists. To this end, we extend state-of-the-art reverse engineering algorithms to deal with Field Programmable Gate Array (FPGA) gate-level netlists equipped with FSM obfuscation. Second, on the basis of realistic reverse engineering capabilities we carefully review the security of state-of-the-art FSM obfuscation schemes. We reveal several generic strategies that bypass allegedly secure FSM obfuscation schemes and we practically demonstrate our attacks for a several of hardware designs, including cryptographic IP cores. Third, we present the design and implementation of *Hardware Nanomites*, a novel obfuscation scheme based on partial dynamic reconfiguration that generically mitigates existing algorithmic reverse engineering.

Keywords: Hardware Reverse Engineering · Hardware Obfuscation · Hardware Nanomites · FSM-based Hardware Obfuscation

1 Introduction

Hardware is the root of trust in virtually any modern computing system [1]. However, modern IC design and fabrication processes are globalized and various (untrusted) stakeholders have access to the designer's valuable hardware IP and are able to commit piracy. Due to piracy it is estimated that IC companies face losses in the range of a billion US dollars in global revenue [2]. Incomplete operative, low-quality counterfeits or malicious manipulations of the underlying hardware can have catastrophic consequences for the security and safety of target systems. Since the threat potential of IC piracy is an increasing concern for practical applications [3], numerous hardware protection schemes have been proposed, see Shakya et al. [4] for a comprehensive survey. Two aspects are mainly addressed: (1) IP theft protection, and (2) reverse engineering protection *a.k.a.* obfuscation. In both cases the goal is to prevent adversaries from gathering high-level information about valuable IP.

Most common strategies to realize hardware protection at the Register Transfer Level (RTL) or gate-level focuses on FSMs [5, 6, 7, 8, 9, 10, 11], since virtually all digital systems use FSMs to define their behaviors. Thus, FSM obfuscation is favored as an approach for concealing overall internal functionality. However, the realistic capabilities of reverse engineering are often neglected in the security analysis of these schemes, hence their security can be limited. To be fair, public information on reverse engineering capabilities has attracted little scrutiny from the scientific community, as discussed in Section 2.2. Little is known about what kind of information can be reverse engineered in an automatic manner, and what kind of information is most challenging to analyze. Reverse engineering is not only associated with illegitimate actions, but there are also various reasons for legitimate applications such as failure analysis and the detection of counterfeit products and hardware Trojans [2].

Goals and Contributions. In this paper, we focus on reverse engineering and obfuscation of FSMs in third-party, gate-level netlists. Our goal is to demonstrate the shortcomings of allegedly secure, state-of-the-art obfuscation schemes by (semi-)automatic reverse engineering and manipulation. To this end, we address reverse engineering techniques that deduce high-level information under realistic assumptions. We then carefully review obfuscation schemes and show how their protection can be defeated. Finally, we introduce our novel defense called *Hardware Nanomites* which evades state-of-the-art algorithmic reverse engineering techniques. In summary, our main contributions are:

- **Deobfuscation of FSM Obfuscation Schemes.** We practically demonstrate the (semi-)automated deobfuscation of several allegedly secure FSM-based obfuscation schemes. In concert with realistic reverse engineering capabilities, we provide comprehensive insights into published security metrics and previous (erroneous) assumptions about reverse engineering to serve as an educational basis for future obfuscation designers and implementers.

- **Novel Technique and Comprehensive Evaluation.** We augment state-of-the-art reverse engineering algorithms to disclose high-level FSM information from FPGA gate-level netlists. We show that the algorithm is effective for several hardware designs while keeping analysis times practical.

- **FSM Reverse Engineering Mitigation.** Finally, we present *Hardware Nanomites*, a novel obfuscation method to hinder FSM reverse engineering. We eliminate fundamental starting points for FSM reverse engineering by leveraging partial dynamic FPGA reconfiguration.

2 Background and Related Work

In the following we introduce our threat model and the state-of-the-art in reverse engineering and IP protection techniques.

2.1 Threat Model

We assume an adversary with access to the flattened gate-level netlist who has no a priori knowledge of the design's internal workings. More precisely, the adversary has no information of module hierarchies, synthesis options, or the names of gates and signals. Furthermore, we assume that the design may incorporate an FSM-based obfuscation technique, as detailed in Section 4. The high-level goal of the adversary is to commit IP infringement such as unauthorized redistribution or overproduction of valuable IP. Since the design is equipped with obfuscation, the adversary is forced to *deobfuscate* the design first. The gate-level netlist can be obtained through several means: (1) chip-level or layout reverse engineering [3, 12] in the case of Application Specific Integrated Circuits (ASICs),

(2) bitstream-level reverse engineering [13, 14] in the case of FPGAs, or (3) directly from the (firm / hard) IP provider [15].

Note that our threat model is consistent with prior research on hardware security [4, 8, 9, 15].

2.2 Gate-Level Netlist Reverse Engineering

Hansen et al. [16] pioneered gate-level netlist reverse engineering. They described several best-practices for a human reverse engineer such as the detection of recurrent modules and common library structures. Shi et al. [17] reported a technique to algorithmically extract FSM gates and signals from ASIC gate-level netlists. Later, Shi et al. [18] described a method to extract diverse functional modules from a gate-level netlist via Boolean function analysis. Li et al. [19] developed a technique to match unknown sub-circuits against library components based on pattern mining of simulation traces and model checking. In further work, Li et al. [20] described how word-level structures can be algorithmically uncovered. Subramanyan et al. [21] extended the algorithmic reverse engineering technique arsenal by extracting functional components such as register files or adders. Since functional identification requires that the input signals of the component are in a specific order, a reverse engineer must examine all orderings to find the correct permutation. Gascón et al. [22] addressed this problem with a template-based solution. Meade et al. [23] extended FSM reverse engineering by retrieving the state transition function from ASIC gate-level netlists. Wallat et al. [24] presented insights on offensive reverse engineering aspects such as the removal of watermarks and the weakening of stream cipher implementations. Recently, Fyrbiak et al. [1] developed a general framework to perform reverse engineering and manipulation of gate-level netlists.

While previous solutions offer methods for algorithmic FSM reverse engineering [17, 23], they specifically target non-obfuscated ASIC gate-level netlists. Recently, Meade et al. [25] introduced theoretical attack descriptions on several FSM obfuscation schemes (a subset of our schemes covered in Section 4), however, they did not perform a practical evaluation of the aforementioned attacks. We discuss several limitations of their presented defense strategy in this paper. Our work fills an important gap to provide detailed insights on reverse engineering processes. Moreover, we target schemes that have been claimed to be resistant to reverse engineering.

2.3 Hardware IP Protection

Modern System-on-Chip (SoC) design often involves the use of numerous reusable IP cores to reduce both time-to-market and cost [26]. The economic advantages of IP use are accompanied by increased security risks for both IP owners and consumers. To protect an owner's IP against piracy threats such as cloning and reverse engineering, various solutions have been proposed.

One line of research focuses on obfuscation techniques at various levels to hinder reverse engineering, i.e. register-transfer level, gate-level, and layout-level, see Shakya et al. [4] for a comprehensive survey. Another line of research focusses to facilitate post-manufacturing control of designed IP core summarized under the concept of hardware metering, see Koushanfar [27] for a comprehensive survey. Solutions related to hardware metering focus on tracking and (un)locking ICs and often incorporate obfuscation techniques to conceal the metering functionality or embed it in a way that it is hard to remove.

Even though numerous techniques for IP protection have been proposed, the security of these techniques is often questionable since realistic reverse engineering capabilities are often neglected or not properly discussed. In this work, we address several shortcomings of FSM-based obfuscation/watermarking schemes and propose a novel methodology to mitigate automated deobfuscation.

3 Automated FSM Reverse Engineering

Preliminaries. From a high-level perspective, an FSM is a computational model that can be in exactly one of a finite number of states at any time. An FSM switches state depending on inputs and its current state and generates outputs to control the operations of other units. Two FSM types can be distinguished: the output of a Moore machine depends solely on the current state, and the output of a Mealy machine depends on both the current state and FSM input. To be more precise, we use the notation in Definition 1 throughout the rest of this paper.

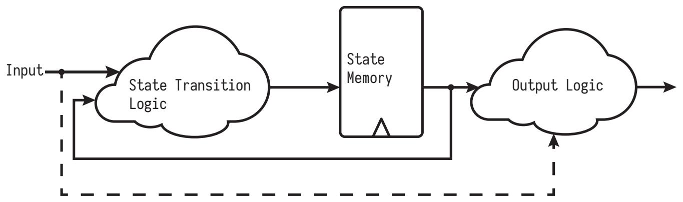


Figure 1: Block diagram of a hardware FSM (dashed line in the case of a Mealy machine).

Definition 1 (Finite State Machine)**.** We define a Finite State Machine by a 6-tuple $(\mathcal{S}, \mathcal{I}, \delta, s_0, \mathcal{O}, \lambda)$: $\mathcal{S}$ is a finite set of states, $\mathcal{I}$ is the input alphabet, $\delta\colon \mathcal{S} \times \mathcal{I} \to \mathcal{S}$ is the state transition function, $s_0 \in \mathcal{S}$ is the initial state, $\mathcal{O}$ is a finite set of output symbols, and λ is the output function ($\lambda\colon \mathcal{S} \to \mathcal{O}$ for a Moore machine, $\lambda\colon \mathcal{S} \times \mathcal{I} \to \mathcal{O}$ for a Mealy machine).

Figure 1 illustrates the high-level structure of an FSM in hardware. An FSM consists of three parts: (1) the state transition logic that implements δ, (2) the memory storing the current state that implements $\mathcal{S}$, and (3) the output logic that implements λ.

State Encoding. Several FSM state encoding styles exist to satisfy diverse optimization goals such as speed or power consumption. Since the encoding affects the hardware implementation (and consequently reverse engineering), we summarize the most common styles:

- **Binary.** Each state is numbered sequentially in order of appearance (starting from 0). Thus, all states can be represented with a $\lceil \log_2(|\mathcal{S}|) \rceil$-bit register. Consequently, the amount of utilized registers is minimized, but the amount of state transition logic is increased.

- **Gray.** Similar to binary state encoding, Gray-encoded states can be represented with a $\lceil \log_2(|\mathcal{S}|) \rceil$-bit register. Based on the employed Gray code, consecutive state values only differ by one bit which reduces the potential for glitches, reduces the amount of combinational logic needed for state transitions, and minimizes power consumption.

- **One-Hot.** Each state is represented with a $|\mathcal{S}|$-bit register). Hence, all register bits except one are equal to 0 at any time. This encoding increases the amount of registers, but simplifies state transition logic in each path to achieve higher clock frequency.

Note that from an obfuscation point of view, binary encodings are preferable over one-hot encodings since the latter grow linearly with the number of states while the former grow logarithmically. While reverse engineering the chosen state encoding, an analyst can gather valuable information about design strategies. For example, power consumption minimization can be assumed in the case of Gray encoding.

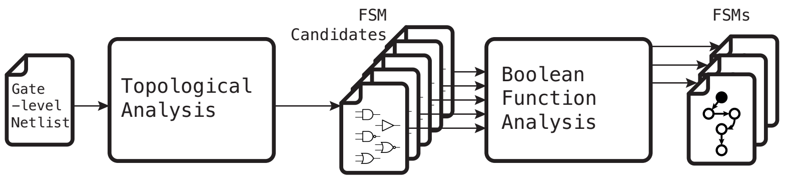


Figure 2: Overview of our FSM reverse engineering work flow. Starting with a third-party, gate-level netlist, we first determine the gates of each FSM candidate using the topological analysis. Afterwards each candidate is processes by the Boolean function analysis to determine the state transition graph.

FSM Reverse Engineering. Figure 2 provides an overview of our two-step reverse engineering technique. First, a topological analysis of the gate-level netlist (Section 3.1) generates a set of FSM candidates consisting of gates and signals that may form an FSM. Second, each candidate is processed by Boolean function analysis (Section 3.2) that determines the set of states $\mathcal{S}$, the input alphabet $\mathcal{I}$, the state transition function δ, and the initial state s_0.

3.1 Phase 1: Topological Analysis

To disclose FSM gates and signals from a gate-level netlist, we transform the netlist into a multi-digraph using an approach similar to the technique described by Shi et al. [18]. Using this format, we analyze the graph topology for FSM characteristics described hereinafter. Even though the block diagram in Figure 1 might appear to be quite elementary, it inherently considers various fundamental properties.

Property I: Registers. Typically FSM state memory registers are controlled by the same set of signals. Therefore, we group all Flip Flops (FFs) with the same clock, enable, and (a)synchronous (re)set signals into a register (represented as set of FF sets) in line ❷ of Algorithm 1. Note that these identified registers are important for further (manual) netlist reverse engineering, since they disclose crucial module-boundary information which partitions the design into easier to analyze functionally-related units.

Property II: Strongly Connected Components. FSM memory and state transition logic form a strongly connected component, i.e. a path exists in each direction between each pair of vertices, see Figure 1. Thus, we first sort out FFs that are not in any strongly connected component with more than 2 vertices ❹ - ❼ since state memory FFs should exhibit a cyclic structure. In addition, we split state registers whose FFs are in different strongly connected components ❽ - ❿ since FSM FFs should influence all other FFs. Note that we are using Tarjan's algorithm to identify strongly connected components [28].

Property III: Combinational Logic Feedback Paths. FSM state memory register output signals possess a feedback to its inputs via a series of combinational gates, forming a *combinational logic feedback path*, see Figure 1. All state memory FFs that do not reach themselves through combinational gates are removed from the register ⓫. In this step, we also determine all state transition logic gates. Therefore, we add all combinational gates in the feedback paths. Subsequently, we augment the set by adding predecessors of all logic gates until we reach a global input or a register gate.

Property IV: Influence/Dependence Metric. FSM candidates with only one FF in the register are rejected to minimize the number of potential FSMs. In addition, we sort out FFs where the intersection of influenced and dependent FFs is smaller or equal to 1 ⓬ - ⓯. More precisely, for each register FF r we determine the set of dependent registers $\mathcal{D}_r$ and the set of influenced registers $\mathcal{I}_r$. We then compute the intersection $\bigcup \mathcal{I}_r \cap \mathcal{D}_r$.

Algorithm 1 Topological Analysis

```
 Input: $D$ - Design netlist
Output: $\mathcal{C}$ - Set of FSM candidates
    // initialization
 1: $\mathcal{C} \leftarrow \emptyset$
    // ensure property I
 2: set of sets $\mathcal{RS} \leftarrow$ registers($D$)
    // ensure property II
 3: set of sets $\mathcal{SCC} \leftarrow$ strongly_connected_components($D$)
 4: for set $register \in \mathcal{RS}$ do
 5:     for gate $g \in register$ do
 6:         if $\mathcal{SSC}$.find_element($g$) == $false$ then
 7:             $register \leftarrow register \setminus \{g\}$
 8: for set $register \in \mathcal{RS}$ do
 9:     if is_splittable($register$) == $true$ then
10:         $\mathcal{RS} \leftarrow (\mathcal{RS} \setminus register) \cup$ split($register$)
    // ensure property III
11: $\mathcal{CLFP} \leftarrow$ combinational_logic_feedback_path($D, \mathcal{RS}$)
    // ensure property IV
12: for set $register \in \mathcal{RS}$ do
13:     for gate $g \in register$ do
14:         if $\bigcup \mathcal{I}_r \cap \mathcal{D}_r \leq 1$ then
15:             $register \leftarrow register \setminus \{g\}$
    // ensure property V
16: for set $register \in \mathcal{RS}$ do
17:     if compute_control_behavior($register$) == 0 then
18:         $\mathcal{RS} \leftarrow \mathcal{RS} \setminus \{register\}$
19:     else
20:         $\mathcal{C} \leftarrow \mathcal{C} \cup \{c(register, \mathcal{CLFP})\}$
21: return $\mathcal{C}$
```

To measure the influence and dependence among all FFs in the register, we compute the mean of all $\bigcup \mathcal{I}_r \cap \mathcal{D}_r$ for each register FF $r \in \mathcal{R}$, normalized by dividing through $|\mathcal{R}|^2$. This metric is used since, in a typical FSM, each state register FF influences and depends on all state register FFs. A value of 1 implies a strong coherence between the FFs whereas a value of 0 implies a loose coherence.

Property V: Control Behavior Metric. Typically, FSM state memory register output signals connect to gates which are not in the strongly connected component (implementing λ). These *control signals* define the circuit's behavior. Since Look-Up Tables (LUTs) are the building blocks used to realize combinational logic in FPGAs, we cannot directly use a previous approach (Shi et al. [18]) which is based on gate type analysis, i.e. whether a control signal connects to the *select* pin of a multiplexer. Moreover, a technique based on the presence of specific gate types is not reliable for netlists equipped with hardware obfuscation.

We solve these issues in a generic way by using a metric to quantify the control behavior ⓱. To this end, we retrieve the FSM output logic gates which are (1) either successors of state memory FFs that are not in the state transition logic, or (2) transition logic gates that connect to gates outside of the strongly connected component. Note that the latter case occurs due to multi-level logic optimization. To measure the control behavior, we approximate the Boolean difference of a state FF output signal that connects to the output logic. We retrieve the minimal Boolean function representation using the Quine McCluskey algorithm [29]. Then, we count how many minimized clauses are affected by the control signal normalized by dividing by the number of clauses, yielding a real value in $[0, 1]$. Note that a value of 1 implies a strong control behavior, whereas a value of 0 implies no control behavior. We remove any candidate which possesses a value of 0 for all control signals. Finally, an FSM candidate is added to $\mathcal{C}$ ⓴.

Use of Metrics for Reverse Engineering. We want to emphasize that (1) control behavior and (2) influence/dependence metrics are especially useful for a human reverse engineer since multiple FSM candidates are typically retrieved. In such cases, the metrics and a detailed report of the topological analysis (e.g,. which FFs influence and depend on each other, or the Boolean functions of control signals) are advantageous for manual analysis, see Section 5.

In summary, a topological analysis determines a set of FSM candidates consisting of register gates, combinational logic gates, and input and control signals that behave similarly to FSMs. Although the analysis discloses relevant gates and logic signals, it does not determine the key elements for reverse engineering of the hardware design, i.e. the state transition function δ. Based on δ, we can deduce the set of (reachable) states $\mathcal{S}$ and analyze the output function λ.

3.2 Phase 2: Boolean Function Analysis

To determine the state transition function δ, the set of states $\mathcal{S}$ and the output function λ for each FSM candidate, we analyze its combinational logic gates with an approach similar to Meade et al. [23]. The key idea is similar to a Breadth-First Search (BFS): we start at the initial state s_0 and for each possible input value we determine the reachable states before moving on to the next level states. Typically, the initial state can be determined from gate configuration values, i.e. initial register values or (re)set signals. To determine the next state from a given current state and input configuration, we evaluate the Boolean functions of the combinational state transition logic. More precisely, each state memory FF data input is represented by a Boolean function whose input variables consist of the state value (FF data output) and FSM input signal values.

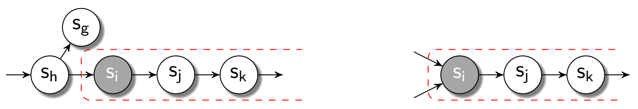
Algorithm 2 shows our technique to retrieve the state transition function δ from an FSM candidate independently of the state encoding. First, we initialize m and $\mathcal{Q}$ with the initial_state determined by the set of registers $\mathcal{R}$ in lines ❶ - ❹. Second, we determine the set of reachable states $\mathcal{S}$ and δ ❺ - ⓬. In line ❼, we iterate through each input signal configuration, e.g., for a 10-bit input signal we enumerate all 2^{10} possible assignments. To compute the evaluate function in line ❽, we use Reduced Ordered Binary Decision Diagrams (BDDs) to represent the Boolean functions. Third, we analyze δ for any *input-independent state register series* and remove candidate state registers that behave like a counter ⓭ - ⓰. Overall, the time complexity is $\mathcal{O}(|\mathcal{S}| \cdot 2^i)$, where i is the bit width across all input signals.

Property: Input Independent State Series. FSM reverse engineering faces several challenges in practice: the similarity of FSMs to counter circuits and non-standard implementation styles by designers. Counters are simplistic FSMs and topological analysis misclassifies them as FSM candidates even though they might not be used for design control. Additionally, counters can be utilized in FSMs even though integrating datapath units into the control path can be considered bad design practice. We provide an example of such an FSM implementation in Listing 5 in the Appendix.

To separate counter registers from state registers, we analyze the state transition function δ. In contrast to FSMs, counters are typically input-independent (except for enable or reset signals). Hence, we search for input-independent state series ⓭. These series start with either a branch or a merge state, cf. Figure 3, and end in a state with

Algorithm 2 Boolean Function Analysis

```
Input: $c \in \mathcal{C}$ - FSM candidate with register $\mathcal{R}$,
    combinational logic gates $\mathcal{L}$, and input signals $\mathcal{IS}$
Output: $m$ - FSM with set of finite states $\mathcal{S}$,
    state transition function $\delta$, and initial state $s_0$
    // initialization
1:  $m.s_0 \leftarrow$ initial_state($c.\mathcal{R}$)
2:  $m.\mathcal{S} \leftarrow \{m.s_0\}$
3:  $\mathcal{Q} \leftarrow Queue()$
4:  $\mathcal{Q}$.enqueue($m.s_0$)
    // determine set of reachable states
5:  while $\mathcal{Q} \neq \emptyset$ do
6:      $s \leftarrow \mathcal{Q}$.dequeue()
7:      for input signal configuration $i \in c.\mathcal{IS}$ do
8:          $s_{new} \leftarrow$ evaluate($c.\mathcal{L}$, $s$, $i$)
9:          $m.\delta(s, i) \leftarrow s_{new}$
10:         if $s_{new} \notin m.\mathcal{S}$ then
11:             $\mathcal{Q}$.enqueue($s_{new}$)
12:             $m.\mathcal{S} \leftarrow m.\mathcal{S} \cup \{s_{new}\}$
    // determine input independent state registers
13: $\mathcal{IIS} \leftarrow$ input_independent_state_series($m$)
14: for gate $r \in c.\mathcal{R}$ do
15:     if has_constant_value($r$, $\mathcal{IIS}$) = $false$ then
16:         $c.\mathcal{R} \leftarrow c.\mathcal{R} \setminus \{r\}$
17: return $m$
```



(a) Branch. (b) Merge.

Figure 3: Input independent state series starting point s_i (series marked in dashed red): (a) Branch: s_i has one successor and one predecessor which is a branch (multiple successors). (b) Merge: s_i has one successor and multiple predecessors.

more than one successor. For each register in the series only the counter registers toggle, hence we remove them in line 16.

Differences to Related Work. As noted earlier in this section, our FSM reverse engineering algorithms are based on previous work by Shi et al. [17] and Meade et al. [30]. Overall, the structure and properties of our algorithms (phases 1 and 2) are similar to both works. However, both previous works specifically target ASIC gate-level netlists, and we had to augment them to work with FPGAs and improve their reliability using Boolean function analysis. Neither work considers FSMs equipped with obfuscation strategies, so we introduced two metrics which aid the human reverse engineer during manual inspection. Also, the separation of counters and FSM circuity was not tackled by the previous approaches.

4 Reverse Engineering and Deobfuscation of FSM Obfuscation Schemes

On the basis of how FSM circuity can be (semi-)automatically reverse engineered, we now analyze several FSM obfuscation schemes and review their claimed security with a particular focus on realistic reverse engineering and manipulation capabilities (Section 4.1 - Section 4.4). We then summarize the different issues in Section 4.5 to serve as an educational basis for future obfuscation designers and implementers.

4.1 HARPOON [8]

HARPOON is a design methodology to obfuscate FSMs and provide a form of authenticity which was utilized in a series of works [5,6,7,8]. The *HARPOON* threat model is equivalent to ours, see Section 2.1.

4.1.1 Design Principle

In general, *HARPOON* augments an FSM with a series of states that form a preceding *obfuscation mode* and an *authentication mode*, see Figure 4. More precisely, the *obfuscation mode* consists of several states $s_0^O, \ldots, s_l^O$ which have to be traversed in a suitable sequence to reach the original initial state s_0 so that the FSM operates as intended. The new initial state of the obfuscated FSM is s_0^O. Note that the input sequence $(i_0, \ldots, i_m)$ required to perform the correct transitions leading to s_0 is called the *enabling key* and it is only known by honest parties. Without knowledge of this key, it should be challenging for an adversary to sell and enable unauthorized design copies. Authentication mode consists of several states $s_0^A, \ldots, s_n^A$ for which the output function λ generates outputs serving as watermarks. Similar to the obfuscation mode, the input sequence required to traverse the authentication states is called the *authentication key*. To deter simulation-based reverse engineering of the design, *HARPOON* leverages *modification cells* so that the design performs incorrect calculations when the FSM is not in post-validation operating mode.

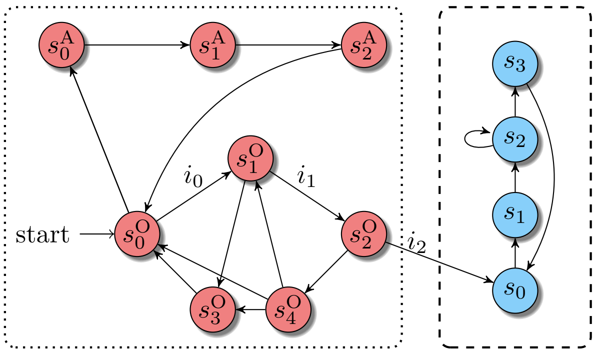


Figure 4: *HARPOON* design methodology example. The original FSM (dashed blue part) is augmented by an obfuscation mode s_0^O, s_1^O, s_2^O, s_3^O, s_4^O and an authentication mode s_0^A, s_1^A, s_2^A. The *enabling key* to reach the original initial state s_0 is (i_0, i_1, i_2).

4.1.2 Security Analysis

Chakraborty et al. assessed the security of *HARPOON* with a "*purely random approach*" [8] (p. 1497) similar to a brute-force attack of the enabling key. This attack does not reflect a realistic adversarial proceeding. We found several strategies to enable unauthorized design activation including: (1) disclosure of the enabling key, and (2) initial state / watermark patching. We acknowledge that a similar attack strategy for enabling key disclosure based on Tarjan's algorithm has been described in a theoretical sense by Meade et al. [25] without experimentation.

For both strategies mentioned above, we must reverse engineer the state transition function δ of the FSM using our aforementioned method in Section 3. Note that Chakraborty et al. [8] claimed a security level of 10^{-47} for 30 state memory FFs and 4 FSM inputs. Hence, our Boolean function analysis determines the state transition function after 2^{34} steps (worst-case).

Disclosure of the Enabling Key. To disclose the enabling key $(i_0, \ldots, i_m)$ from a gate-level netlist, the state transition function δ is analyzed using the state transition graph. An important observation is that there is no path from the original $s_0, s_1, \ldots$ FSM states back to the preceding states in obfuscation mode, see Figure 4. Additionally, the state transition function of the original FSM typically consists of a cyclic structure and thus forms a strongly connected component which can be identified with Tarjan's algorithm [28]. Subsequently, we can disclose the enabling key $(i_0, \ldots, i_m)$ by examining which inputs lead to the original initial state s_0 (e.g., by using Dijkstra's shortest path algorithm).

Initial State Patching. Based on the observations for enabling key disclosure, we can also patch the state memory to entirely skip the obfuscated mode. Therefore, we have to alter initial values of the FFs to s_0 (derived by δ and the strongly connected component property). For Xilinx FPGAs, FFs and latches include an initialization attribute `INIT` which sets the initial values of state outputs after configuration [31]. For other gate libraries, the (a)synchronous (re)set signals may be rerouted to *GND* or *VCC* depending on s_0. For ASICs, typical FFs in gate libraries offer `Q` and `QN` (negated `Q`) output pins, so these signals can be multiplexed on reset to model either a logic 0 or 1 for the state transition function.

Watermark Manipulation. Similar to initial state patching, we can manipulate the watermark to invalidate the design's authenticity and survive post-silicon authentication where scan-FFs can be used to set the state memory to s_0^{A}. Therefore, we must alter the output function λ, so that the output values are changed for authentication states $s_0^{\mathrm{A}}, \ldots, s_n^{\mathrm{A}}$ (e.g., by negating each output). For Xilinx FPGAs, output logic is typically implemented in LUTs so we can simply change its `INIT` value to alter its functionality. For other gate libraries (e.g., of ASICs), we may add additional inverter gates to alter the functionality and thus the watermark.

4.2 Dynamic State Deflection [32]

Dynamic State Deflection is an FSM obfuscation technique that is used to prevent the unauthorized overwriting of the FSM state memory register. Our proposed *initial state patching* uses this type of overwriting. The threat model employed by Dofe et al. [32,33] is equivalent to our threat model, see Section 2.1.

4.2.1 Design Principle

The general principle of *Dynamic State Deflection* is to protect each original FSM state to verify whether the correct *enabling key* i_k is present for each state transition. In case any invalid key $i \neq i_k$ is present, the modified state transition δ yields a deflection to so-called *black hole clusters* $s_{b0}, \ldots, s_{bn}$, see Figure 5, and thus protects from overwriting of the FSM state memory register. A key feature of its construction is that once an invalid key $i \neq i_k$ is assigned to the design, it never reaches an original state. Since each state transition verifies the presence of a valid enabling key part, the FSM has to be augmented with a dedicated enabling key port. Note that the scheme also builds up on existing techniques such as *HARPOON* [8] to protect the design with a preceding obfuscation mode and enabling key $(i_0, \ldots, i_m, i_k)$. To be more precise, the $(i_0, \ldots, i_m)$ part refers to the enabling key of the preceding obfuscation mode, while the latter i_k refers to the key validated for every original state transition.

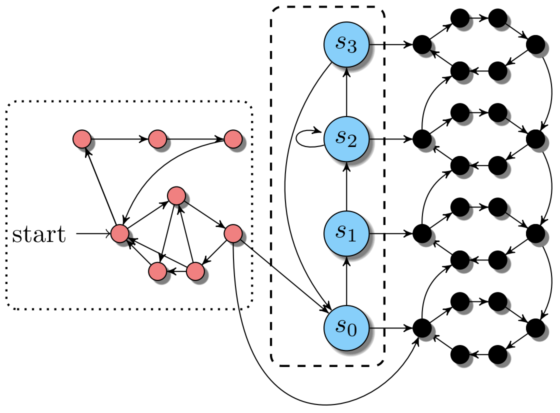


Figure 5: *Dynamic State Deflection* design methodology example. The original FSM (dashed blue part) is augmented by an *HARPOON* obfuscation mode (dotted red part) and each original state is protected by a black hole (states marked in black).

4.2.2 Security Analysis

Dofe et al. examined the security for the *Interlocking Obfuscation* scheme and claimed to raise the bar for reverse engineering since the transition between states is based on a complex structure. However, we show an automated strategy to enable unauthorized design activation by disclosure of the enabling key. Although a similar attack strategy based on Tarjan's algorithm for key recovery has been described theoretically by Meade et al. [25], we provide a practical implementation. For this strategy, we have to reverse engineer the state transition function of the FSM using our aforementioned method in Section 3. Note that Dofe et al. [32] used a 12-bit enabling key size for their evaluation. Our Boolean function analysis is able to practically determine the state transition function in a similar context.

Disclosure of the Enabling Key. Similar to *HARPOON*, we exploit the characteristic construction of the state transition function δ to retrieve the enabling key $(i_0, \ldots, i_m, i_k)$ from the gate-level netlist. In particular, the black hole clusters (forming a strongly connected component) can be distinguished from original states using Tarjan's algorithm [28], see Figure 5. Note that the original FSM states do not form a strongly connected component with the black hole clusters, since there exists no path from the black hole clusters back to any original state. After distinguishing original from obfuscation mode states, we can analyze δ for the inputs leading to s_0 (e.g., by using Dijkstra's shortest path algorithm).

On State Transition Function Patching. To increase security, Dofe et al. proposed increasing the key size for i_k (e.g., to 64-bits). The key is checked prior to an original state transition. Even though this key size increase appears to improve protection against a naive brute-force of δ, a synthesizer will implement a characteristic comparator that can be automatically identified and manipulated, especially for large key sizes [1]. Note that after reverse engineering the location of the comparator circuit, we may simply patch the state transition function δ (e.g., to report *true* for each value of i_k) and ultimately re-enable initial state patching. For Xilinx FPGAs, state transition function logic is typically realized in LUTs so we can simply change its `INIT` values to alter the functionality and report *true* for each i_k input. For ASICs, we may insert or remove certain combinational logic gates to alter the functionality of state transition function logic accordingly. Hence, either the key can be reverse engineered by a naive brute-force evaluation of δ for smaller key sizes or an in-depth analysis of comparator circuits that are present for larger key sizes can be used if a brute-force evaluation might not be efficient (e.g., $\geq 2^{40}$ on a standard computer).

4.3 Active Hardware Metering [9]

Hardware Metering [27, 34, 35] refers to a collection of security mechanisms and protocols to facilitate post-manufacturing control of designed IP cores. The collection of metering techniques can be broadly separated into (1) *passive*, and (2) *active* techniques. Passive metering facilitates unique chip identification, whereas active metering additionally enables designers to (un)lock chips. We focus on active hardware metering. Note that the threat model for active hardware metering is equivalent to our threat model, see Section 2.1.

4.3.1 Design Principle

Active hardware metering augments an original FSM with preceding states which must be traversed in the correct order to reach the original initial state s_0. To this end, the original state register with s FFs is augmented by l FFs, which results in 2^l additional states $s_0^{\mathrm{O}}, \ldots, s_{2^l-1}^{\mathrm{O}}$ for a binary-like state encoding, see Section 3. In particular, the new initial state s_0^{O} is determined by a device-unique and unpredictable Random Unique Block (RUB), see Figure 6. The state transition function δ is generated using one or more small ring counters (e.g., 16 or 32 states) which are then modified by randomly reconnecting and adding several edges. These small ring counters are then combined to form the obfuscated FSM and original states are also randomly connected to additional black hole states similar to the structure explained for *Dynamic State Deflection* in Section 4.2. To unlock a design, the initial register value of s_0^{O} is read out by the user and sent to the IP provider who determines the enabling key $(i_0, \ldots, i_m)$. Without the enabling key it should be challenging for an adversary to sell and enable unauthorized copies of the design.

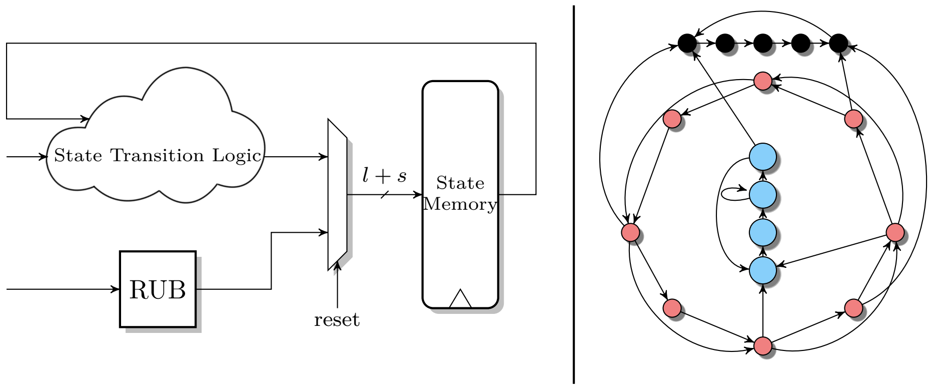


Figure 6: *Active Hardware Metering* technique example. The RUB response determines the initial state of the FSM. The enabling key then determines the transition from the obfuscation mode states (marked in red) to the original initial state of the FSM (marked in blue).

4.3.2 Security Analysis

The security of active hardware metering was previously examined for reverse engineering and manipulation, i.e. *control signal capture and replay*[1]. To increase security, the authors propose to alter the FSM so that the reset state is RUB-dependent as well, hence an FSM only operates correctly when it receives a specific stream of signals from the RUB after reset. Thus, the authors conclude that this renders reverse engineering "*much more difficult*" and the control signal capture and replay "*almost impossible*". Koushanfar [10] proved the security of the scheme with a related structure. Implementation details are dedicated to the state transition function δ, but the output function λ is hardly considered. It is noted that "*the BFSM inputs are Primary Inputs (PI) and its outputs are Primary Outputs (PO) since they are the same as the PI and PO in the original design*" [10] (p. 57). Despite these claims and associated proof, we now provide an attack to perform unauthorized design activation by means of initial state patching.

Initial State Patching. To disclose the original initial state s_0 from the gate-level netlist, we carefully analyze the output generation logic λ. An important observation is that λ is only affected for original states by construction of *Active Hardware Metering* and thus we can infer original states by analysis of the Boolean functions for the output logic gates. To be more precise, the output logic will typically implement control signals that only become active (e.g., logical `'1'`) only for specific states. Based on such comparator circuitry we can directly read out original state configurations from the Boolean functions of the control signals. For example, a comparator circuit is added to the output logic λ to ensure that the l obfuscation FFs hold a pre-defined value (e.g, zero) to safeguard correct functionality of the design. Note that large comparators can be also automatically identified due to its characteristic functionality [1].
We then initialize Boolean function analysis with the original states extracted from output logic λ. Due to the construction of *Active Hardware Metering*, we can only transit from an original state to either other original states or black hole states. Therefore, the complexity of our Boolean function analysis is not 2^{s+l} for an $l + s$-bit state register but rather bound by the (usually linear) number of original and black hole cluster states. Similar to *Dynamic State Deflection* in Section 4.2, we can automatically distinguish between original states and black hole states using Tarjan's algorithm as the black hole states from a strongly connected component in the state transition graph. In contrast to *HARPOON* and *Dynamic State Deflection*, we cannot directly read the original initial state s_0 of the state transition graph in each case. Nevertheless, typical designs implement a *reset* state which initializes the data path registers and thus, by analysis of which control signal causes such a characteristic *reset* behavior, we can recover the original initial state s_0. Otherwise the analyst has to reverse engineer (parts of) the datapath to identify an initial state that makes sense.

We want to emphasize that recovery of the original initial state s_0 by the aforementioned *reset* behavior can be performed independently of the Boolean function analysis. Thus, neither a large (e.g., > 64) number of input signals nor a large number of state memory FFs prevent initial state patching.

On Enabling Key Disclosure. We want to highlight that for a small number of states $|\mathcal{S}|$ and a number of input signals i, an enabling key disclosure is possible. Since the initial value of the state memory FFs (defined by the RUB) can be read out by the adversary, he is able to perform Boolean function analysis of the FSM circuit. For example, Alkabani et al. concluded that a brute-force attack on FSMs with up to 18 FFs and 8 input signals does not yield success, cf. Table 3 in [9], however, targeted reverse engineering of the state transition function δ is possible for these values (2^{26} in the worst-case which

[1]"*In this attack, Bob* [the adversary] *attempts to bypass the FSM by learning the control signals and attempting to emulate them. Bob may completely bypass the FSM by creating a new FSM that provides control signals to all functional units, and control logic (e.g. MUXs and FFs) in the datapath.*" [9] (p. 299)

takes a couple of minutes to perform on commodity hardware, see Section 5). It is possible to analyze δ to identify the original FSM states by investigating which states affect the output function λ and control other parts of the circuit. In further work, Koushanfar [35] evaluated this technique for 20 FFs and 64 input signals. Enabling key disclosure would not be effective for such designs, but initial state patching may be performed.

In related work, Gören et al. [36] proposed a partial bitstream protection scheme based on obfuscation, partial reconfiguration, and Physical Unclonable Functions (PUFs). This scheme leverages partial bitstreams to generate PUF responses to steer obfuscated FSM behavior. However, as no original FSM state may transit back to an obfuscation state, we can exploit control signal characteristics (similar to *Active Hardware Metering*) to determine the original initial state.

4.4 Interlocking Obfuscation [37]

Interlocking Obfuscation is an FSM obfuscation technique that is used to provide anti-tamper hardware [37]. The threat model used by Desai et al. is equivalent to our threat model, see Section 2.1.

4.4.1 Design Principle

Similar to *HARPOON*, the *Interlocking Obfuscation* scheme augments the original FSM with an obfuscation mode $s_0^{\text{O}}, \ldots, s_l^{\text{O}}$ and a *code-word*, see Figure 7. The code-word is interwoven with the state transition function δ, so that δ is not only dependent on the current state and input but also on the value of the code-word. Moreover, δ modifies the code-word, i.e. $\delta\colon \mathcal{S} \times \mathcal{I} \times \mathcal{C} \to \mathcal{S} \times \mathcal{C}$ for the set of possible code-words $\mathcal{C}$. Hence, without knowledge of the initial correct code-word value $c_0 \in \mathcal{C}$ (which is only available to honest parties) it should be challenging for an adversary to unlock and tamper with the design.

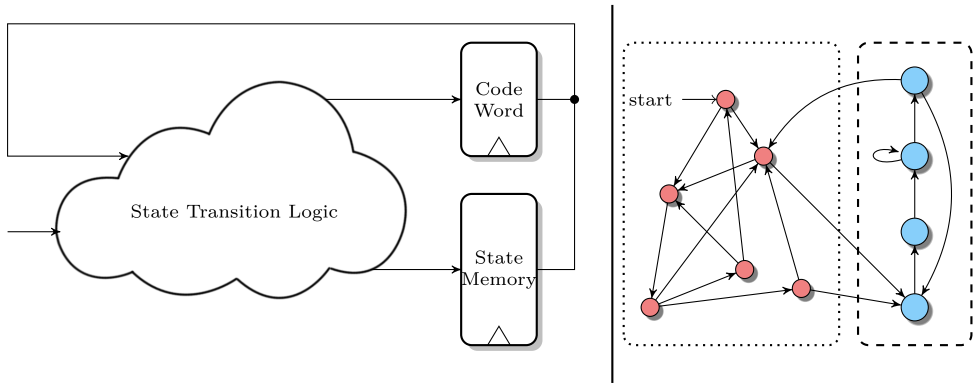


Figure 7: *Interlocking Obfuscation* design methodology example. The original FSM (dashed blue part) is augmented by an obfuscation mode and a code-word (dotted red part).

4.4.2 Security Analysis

The security of *Interlocking Obfuscation* was assessed with a brute-force approach [37] where the adversary tries all combinations to separate the code-word from the actual state memory FFs and subsequently find the correct initial code-word c_0. Since Desai et al. choose 56 FFs in their evaluation (8 state memory + 48 code-word), the number of combinations is $\binom{56}{8} \cdot 2^{48} \approx 2^{78}$ (in case the adversary knows the number of states) and therefore is impractical to compute. Before we detail two generic problems with the *Interlocking Obfuscation* scheme, we note that Meade et al. [25] theoretically described a key recovery approach, however, their approach requires the state transition graph, necessitating a separation of the code-word and state memory FFs. Meade et al. did not provide information on how to separate the FFs and thus enable the computation of the decisive state transition graph. Thus, their attack would only work for small code-word sizes.

Initial State Patching. A key feature of *Interlocking Obfuscation* is the interwoven structure of the code-word and the state memory FFs which should be challenging to reverse engineer. To patch the initial state, we use a strategy that is similar to the one we used for *Active Hardware Metering*, see Section 4.3. Since the output function λ is not affected by the code-word, we can simply read out FF assignments from the Boolean functions and thus separate code-word and state memory FFs. Since the "code-word is not needed to compute a correct next state for all original state" (p. 3 [37]), we can identify the original initial state s_0 either by *reset* behavior, or Boolean function analysis similar to *Active Hardware Metering*.

On Anti-Tamper Hardware. Another issue is the effectiveness of anti-tamper protection. Even though the FSM is obfuscated and only a valid code-word enables the design, the tamper resistance of this scheme is questionable since it only protects the design control path. Desai et al. [37] chose an Advanced Encryption Standard (AES) design for their evaluation, however, several AES datapath attacks on Sboxes have been published [38, 39] (the first reference appeared several years prior to the publication by Desai et al.). These attacks leak the secret key and can be performed irregardless of any obfuscated control path, rendering the anti-tamper property questionable.

4.5 Lessons Learned

We now summarize the different issues of the schemes to provide future obfuscation designers and implementers with a clear picture of what is and is not currently possible with respect to automatic reverse engineering.

Topological Analysis Discloses FSM Gates. FSMs exhibit several characteristics (e.g., cyclic structure with a combinational logic feedback path), so state memory FFs and transition and output logic gates can be automatically extracted from a gate-level netlist, independent of the number of inputs or state memory FFs.

Separation of Obfuscated Parts. A common denominator of the presented schemes is the possible separation of original and obfuscation circuitry, since the obfuscation circuitry is not logically entangled with remaining logic. For example, the output function λ does not depend on obfuscation states in *Active Hardware Metering* [9] or *Interlocking Obfuscation* [37]. This observation enables separation and ultimately enables key recovery or initial state patching. Bogus output generation for obfuscation states (as realized in *HARPOON* [8]) indeed provides an effective countermeasure against the separation of obfuscation and original FSM circuitry. However, adding the capability to generate a bogus output is not a generic solution (e.g., in case the FSM steers an actuator).

Complexity of Boolean Function Analysis. As noted earlier in this section, the complexity to retrieve the state transition function δ is $\mathcal{O}(|\mathcal{S}| \cdot 2^i)$, where i is the combined bit width of all input signals. If the state space and number of input signals is small, δ can be reverse engineered, yielding potential enabling keys. Moreover, this approach provides the state transition graph which is valuable for manual analysis. Scaling the FSM (e.g., to 2^{20} states) is not trivial since a straightforward implementation will require a large amount of combinational logic for state transitions. Scaling the FSM input signal count (e.g., to 40) may be prohibitive since the signals must be meaningfully connected to external devices.

Eavesdropping on the Enabling Key. *Eavesdropping* on the enabling key may also be a realistic attack vector, however, it requires access to a benign device and a valid enabling key. Moreover, further reverse engineering is required to understand how the enabling key is transferred from a communication interface to the FSM and subsequently processed.

5 Evaluation

We now provide an evaluation of our new automated FSM reverse engineering and manipulation strategies to underline the insecurity of the allegedly secure FSM-based obfuscation strategies presented in Section 4. We first present how we implemented the different obfuscation strategies.

Implementation. Since no openly available implementation of the different obfuscation schemes was available, we generated the gate-level netlists for the selected hardware designs using Xilinx ISE (version 14.7) ourselves. For *HARPOON* we selected 14 additional states and 8-bit enabling key input signals. For *Dynamic State Deflection* we used the *HARPOON* obfuscation mode and added black hole clusters of 5 states for each original state. We deliberately omit an isolated evaluation of *HARPOON* since this technique is already included in *Dynamic State Deflection*. Our selected parameters for *HARPOON* are similar to the original work [8] as the authors choose an FSM with 4 states and 10-bit enabling key size, cf. Section V C [8]. Our parameters for *Dynamic State Deflection* are also similar to the original work [33] that utilized a 12-bit enabling key. We realized *Active Hardware Metering* using 256 additional states and an 8-bit enabling key, as well as a black hole cluster of 5 states per original state. This parameter choice is similar to the original work [9], that evaluated obfuscated hardware designs with 18 FFs and 8-bit input which has a comparable worst-case complexity, see Table 3 [9]. *Interlocking Obfuscation* was generated with a *4*-bit code-word. Our FSM reverse engineering algorithms Algorithm 1 and Algorithm 2 are implemented with the assistance of HAL [1]. As soon as HAL is openly released, we will publish our analysis plugin and (obfuscated) gate-level netlists. We want to note that all evaluated obfuscation strategies are realized with a binary state encoding since results for Gray-encoded FSMs are similar. As explained in the previous section, one-hot state encodings are not reasonable for FSM obfuscation since the number of utilized FFs grows linearly with the number of states yielding a large area overhead.

5.1 Case Study: Cryptographic Designs

Even though most cryptographic primitives in use today are resilient against traditional attacks, adversaries leverage implementation attacks (e.g., side-channel analysis) to undermine vital security goals such as confidentiality and integrity [1]. To counteract such attacks, numerous strategies have been investigated by the research community and industry has yielded highly-optimized and secure implementations [40]. Thus, from an economic point of view cryptographic implementations are valuable IP that are worth protecting from IP infringement (e.g., by using FSM obfuscation).

For this case study, we selected two cryptographic hardware designs: (1) an iterative AES IP core, and (2) an iterative Secure Hash Algorithm (SHA)-3 IP core, since both cryptographic building blocks are widely deployed in practice. State transition graphs of both FSMs are depicted in Figure 14 in the Appendix. We obfuscated each hardware design with *Dynamic State Deflection*, *Active Hardware Metering*, and *Interlocking Obfuscation* as described in the previous section.

5.1.1 HARPOON and Dynamic State Deflection

AES. Our topological analysis identifies 2 FSM circuits after about 2 minutes of computation time on a standard laptop:

(1) 32 FFs, 131 input signals, control value 0.971, influence/dependence value 0.625

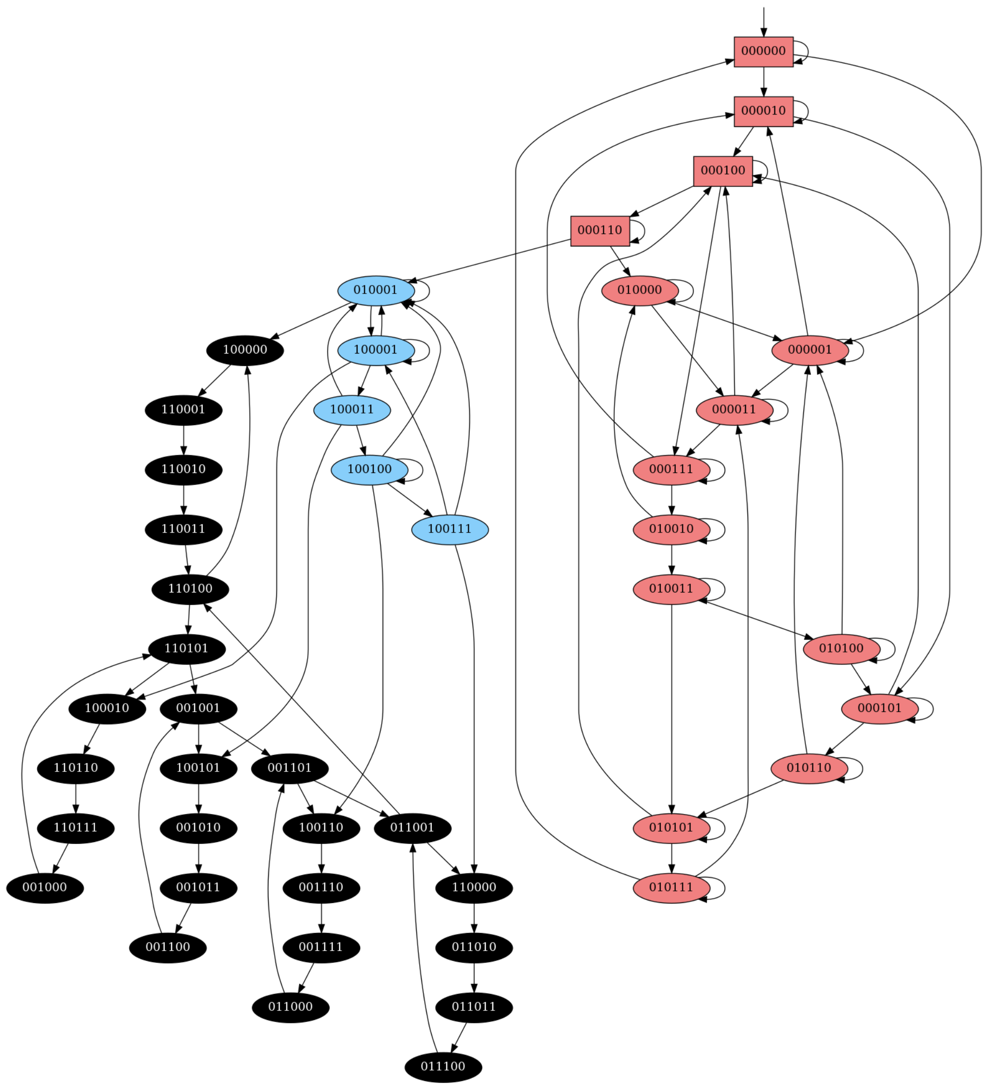


Figure 8: State transition graph of AES IP core obfuscated with *Dynamic State Deflection* (including *HARPOON*). Tarjan's algorithm splits all states into 3 strongly connected components: obfuscation mode of *HARPOON* (marked in red), black hole cluster states (marked in black), and original FSM (marked in blue). Rectangle nodes mark the sequence from initial state 000000 to the original initial state 010001. Input values for each state transition are deliberately left out for readability.

(2) 8 FFs, 21 input signals, control value 0.519, influence/dependence value 0.625

Our analysis indicates that each FF in the first candidate only connects to 4 successor gates (an unlikely scenario for control signals that steer a complex data path) and there is no FF subset where all FFs depend on each other. Based on these properties it seems unlikely that this candidate implements an FSM and quick manual analysis reveals that this circuit actually implements a shift-register. Since the influence/dependence value of the second candidate is $0.625 \neq 1.0$, we analyze the topological analysis report and see that the FFs form two groups where all FFs influence and depend on each other: one group with 2 FFs and the other one with 6 FFs, see Listing 1. We omit the group with

2 FFs since we are searching for an obfuscated FSM with >> 4 states. Boolean function analysis of the 6 FFs yields the state transition graph shown in Figure 8. As described in Section 4.2, an analysis of the strongly-connected components yields three parts: the obfuscation mode of *HARPOON*, the black hole clusters of *Dynamic State Deflection*, and the original FSM. With the state transition function δ we can generate the enabling key by searching for a path from the initial state to the original initial state (marked as red boxes in Figure 8). Similarly, we can perform an initial state patching attack by altering the `INIT` value of each state memory FF accordingly. The 2^{22} (6 FFs and 16 input signals) computations for the Boolean function analysis took about 5 min on a standard laptop.

Listing 1: Excerpt of the topological analysis report of AES IP core obfuscated with *Dynamic State Deflection* (including *HARPOON*). Gate names have been blinded so that no information can be inferred by names. FFs U1, U2, ... U8 form 2 distinct groups where all FFs influence and depend on each other.

```
...
[+] U1 influences and depends on: U1 , U2 , U3 , U4 , U5 , U6
[+] U2 influences and depends on: U1 , U2 , U3 , U4 , U5 , U6
[+] U3 influences and depends on: U1 , U2 , U3 , U4 , U5 , U6
[+] U4 influences and depends on: U1 , U2 , U3 , U4 , U5 , U6
[+] U5 influences and depends on: U1 , U2 , U3 , U4 , U5 , U6
[+] U6 influences and depends on: U1 , U2 , U3 , U4 , U5 , U6
[+] U7 influences and depends on: U7 , U8
[+] U8 influences and depends on: U7 , U8
...
```

SHA-3. Results for the SHA-3 IP core are similar to those from the AES core. Our topological analysis detects 2 FSM candidates after around 24 minutes on a standard laptop:

(1) 128 FFs, 2573 input signals, control value 0.803, influence/dependence value 0.016

(2) 6 FFs, 19 input signals, control value 0.408, and influence/dependence value 1.0

The first candidate belongs to the iterative SHA-3 data path and does not implement an FSM as indicated by the low influence/dependence value. The latter candidate has characteristics of an FSM circuit. Boolean function analysis of this candidate yields a state transition graph that is similar to the one generated for AES, shown in Figure 8. The graph leads to the disclosure of the enabling key and enables initial state patching. The 2^{25} (6 FFs and 19 input signals) computations for the Boolean function analysis took about 27 minutes on a standard laptop.

For both obfuscated hardware designs (AES and SHA-3), our input independent state series analysis reports that all FSM FFs yield non-state behavior (triggered by the black hole clusters). Hence, we can use this information to distinguish whether the FSM incorporates black hole state clusters if they are implemented with input-independent state transitions.

5.1.2 Active Hardware Metering and Interlocking Obfuscation

AES. Our topological analysis identifies 3 FSM candidates after about 4 minutes of computation on a standard laptop:

(1) 32 FFs, 131 input signals, control value 0.971, influence/dependence value 0.625

(2) 2 FFs, 7 input signals, control value 0.463, influence/dependence value 1.000

(3) 15 FFs, 17 input signals, control value 0.845, influence/dependence value 0.742

As the first candidate does not implement an FSM, see Section 5.1.1, and the second candidate is too small to implement any obfuscated FSM, we focus on the third FSM candidate. Since the influence/dependence value is $0.742 \neq 1.0$, the topological analysis report is needed to examine the influence/dependence of each FF. Analogous to Listing 1, we can exclude 6 FF since they do not form a subgroup where all FFs influence and depend on each other. As a result, we focus on the remaining 9 FFs of the third candidate.

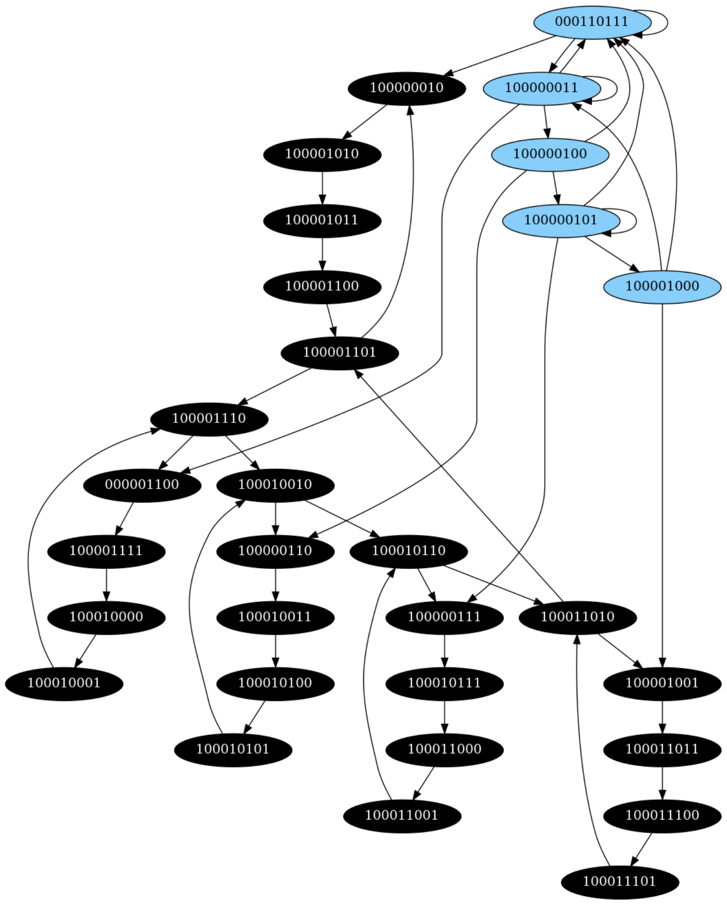


Figure 9: State transition graph of AES IP core obfuscated with *Active Hardware Metering*. Tarjan's algorithm splits all states into 2 strongly connected components: black hole cluster states (marked in black), and original FSM states (marked in blue). Input values for each state transition are deliberately left out for readability.

Listing 2: Excerpt of the topological analysis report of AES IP core obfuscated with *Active Hardware Metering*. Gate and net names have been blinded so that no information can be inferred by names. FFs $U1$, $U2$, $\ldots U9$ connect to control signals $O1$ and $O2$. Boolean functions of $O1$, $O2$, $O3$ are determined by Quine McCluskey Boolean function minimization, `~` refers to a negated literal, `+` to a disjunction, and `*` to a conjunction. Based on the Boolean functions, we infer two states where we assign each FF the Boolean value in the formula.

```
...
[Control Signal] O1 =  U1*~U2*~U3*~U4*~U5*~U6* U7*~U8* U9
[Control Signal] O2 =  U1*~U2*~U3*~U4*~U5* U6*~U7*~U8*~U9
...
```

Listing 2 shows an excerpt of the topological analysis report of the third candidate. The characteristic control signals $O1$ and $O2$ only depend on the 9 FFs (and not on the 6 FFs excluded via influence/dependence analysis). This information confirms that the 6 FFs do not belong to the FSM state memory. In the listing, we minimized Boolean functions of 2 control signals and we can infer 2 potential states, namely $s_1 = U1 \ldots U9 = 100000101$, and $s_2 = U1 \ldots U9 = 100001000$ which we feed to the Boolean function analysis yielding the state transition graph in Figure 9 (similar to the adapted original in the Appendix). Note that the computation time for the Boolean function analysis took around 4 minutes on a standard laptop. Based on the state transition graph, we deduce the original initial state $s_0 = 000110111$ as it behaves as a *fallback* state (in case a specific condition holds) and the other states form a strongly connected component (in case this specific condition does not hold).

As explained in Section 4.4, an attack on the *Interlocking Obfuscation* scheme is the same as for the *Active Hardware Metering* scheme and thus we omit the results for the former approach.

SHA-3. The results for the SHA-3 IP core are similar to those found for AES in the previous section. Our topological analysis detects 2 FSM candidates after about 38 minutes of computation on a standard laptop:

(1) 128 FFs, 2574 input signals, control value 0.250, influence/dependence value 0.016

(2) 9 FFs, 19 input signals, control value 0.406, influence/dependence value 1.000

Similar to the results for *Dynamic State Deflection*, the first candidate belongs to the iterative SHA-3 data path while the latter candidate shows characteristics of an FSM circuit.

Listing 3: Excerpt of the topological analysis report of SHA-3 IP core obfuscated with *Active Hardware Metering*. Gate and net names have been blinded so that no information can be inferred by names. FFs $U1$, $U2$, $\ldots U9$ connect to control signals $O1$, $O2$, and $O3$. Boolean functions of $O1$, $O2$, and $O3$ are determined by Quine McCluskey Boolean function minimization, `~` refers to a negated literal, `+` to a disjunction, and `*` to a conjunction. Based on the Boolean functions, we infer 3 potential states where we assign each FF the Boolean value in the formula and all other not considered FFs (e.g, $U2$ in $O1$) are assigned logical `'0'`.

```
...
[Control Signal] O1 =  U1*            ~U5* U6*~U7*~U8*~U9
[Control Signal] O2 =  U1*            ~U5*~U6* U7*    ~U9
[Control Signal] O3 =     ~U2*~U3* U4* U5*~U6* U7* U8* U9
...
```

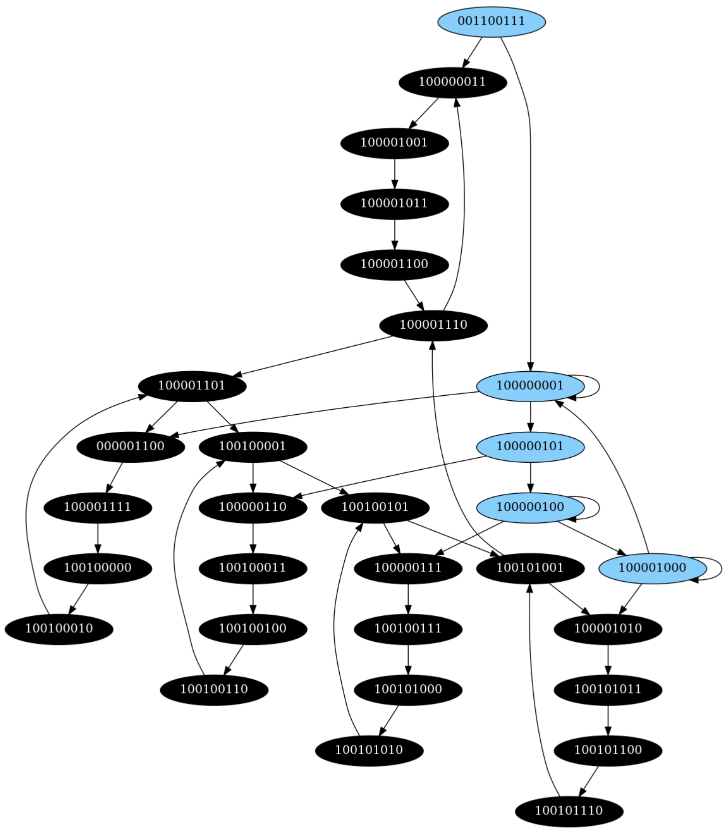


Figure 10: State transition graph of of SHA-3 IP core obfuscated with *Active Hardware Metering*. Tarjan's algorithm splits all states into 3 strongly connected components: black hole cluster states (marked in black), original reset state (marked in blue), and original FSM states (marked in blue). Input values for each state transition are deliberately left out for readability.

Listing 3 shows an excerpt of the topological analysis report of the second FSM candidate. We see the minimized Boolean functions of 3 control signals $O1$, $O2$ and $O3$. Each signal implements characteristic control behavior since each becomes a logical `'1'` under one specific state memory configuration. Based on these 3 signals, we infer 3 potential states namely $s_1 = U1 \dots U9 = 100001000$, $s_2 = U1 \dots U9 = 100000100$, and $s_3 = U1 \dots U9 = 000110111$ and feed them to Boolean function analysis which yields the state transition graph in Figure 10. We want to highlight that the state values in Figure 10 are not in order $U1 \dots U9$ and thus differ from $s_1, \dots, s_3$. As described in Section 4.3, Boolean function analysis is not dependent on the number of FFs but rather on the number of original states due to the specific construction of the *Active Hardware Metering* scheme. Original states transit either to other original states or to black hole cluster states. The computation for the Boolean function analysis took around 17.5 minutes on an standard laptop. Based on the state transition graph, we deduced the original initial state $s = 001100111$ to perform initial state patching of obfuscation and state memory FFs. Initial state patching to the original reset state s is also possible without Boolean function analysis. Control signal $O3$ implements a data path reset functionality since it only becomes active in the initial state and connects to the reset ports of 1608 FFs (1600 SHA-3 data path FFs and 8-bit Linear Feedback Shift Register (LFSR) FFs).

As explained in Section 4.4, an attack on the *Interlocking Obfuscation* scheme is the same as one on the *Active Hardware Metering* scheme and thus we omit the results for the former approach.

5.2 Case Study: Communication Interfaces

Virtually all hardware designs implement dedicated communication interfaces to exchange information with peripheral devices. These interfaces range from simplistic parallel interfaces to complex serial interfaces. Since communication interfaces receive potentially untrusted data from peripheral devices, design activation measures implemented in these building blocks can be used to reject untrusted data before it is forwarded to internal hardware modules, thus protecting valuable IP.

For this case study, we selected a serial Universal Asynchronous Receiver Transmitter (UART) interface since this interface is widely deployed in real-world embedded systems. We augmented the UART with our previously-described FSM obfuscation schemes and evaluated each approach separately.

5.2.1 HARPOON and Dynamic State Deflection

UART. Our topological analysis detects 5 FSM candidates after about 17 seconds of computation time on a standard laptop:

(1) 6 FFs, 8 input signals, control value: 1.0, influence/dependence value: 1.0

(2) 16 FFs, 6 input signals, control value: 0.500, influence/dependence value: 0.445

(3) 4 FFs, 18 input signals, control value: 0.400, influence/dependence value: 1.000

(4) 17 FFs, 6 input signals, control value: 0.785, influence/dependence value: 0.495

(5) 5 FFs, 14 input signals, control value: 0.698, influence/dependence value: 0.520

Based on the high control and influence/dependence values, we identify the first candidate as a potential FSM. Boolean function analysis of the 6 FFs and 8 input signals yields a state transition graph similar to the one depicted in Figure 8. Analogously, Tarjan's algorithm splits the states into three sets (obfuscation mode states, black hole states, and original states) and thus enables disclosure of the enabling key and initial state patching. The 2^{14} (6 FFs and 8 input signals) computations for Boolean function analysis took about 2 s on a standard laptop.

We now briefly describe the other four identified circuits and why they are marked as potential FSMs by topological analysis. The second candidate represents the transmission FSM with (2 FFs), a clock divider (8 FFs) and a countdown counter (6 FFs). Based on the topological analysis report, we can separate the transmission FSM FFs from the datapath FFs since each of the 16 FFs depends on the 2 FSM state memory FFs. The third candidate refers to a 4-bit counter which naturally exhibits a high internal influence/dependence but a low overall controllability since counters and FSMs behave similarly. Analogous to the transmission FSM, the fourth candidate refers to the receiving FSM (3 FFs) which includes a clock divider (8 FFs) and a countdown counter (6 FFs). The last candidate refers to the state machines for the empty/full logic of both receiving and transmission FIFO circuits used in the design. Each candidate refers (partially) to genuine FSM circuits.

5.2.2 Active Hardware Metering and Interlocking Obfuscation

UART. Similar to the results for *Dynamic State Deflection*, our topological analysis detects 5 FSM candidates:

(1) 9 FFs, 8 input signals, control value: 1.000, influence/dependence value: 1.000

(2) 16 FFs, 6 input signals, control value: 0.500, influence/dependence value: 0.445

(3) 4 FFs, 18 input signals, control value: 0.400, influence/dependence value: 1.000

(4) 17 FFs, 6 input signals, control value: 0.785, influence/dependence value: 0.495

(5) 5 FFs, 14 input signals, control value: 0.698, influence/dependence value: 0.520

Since the control and influence/dependence values are 1.0, we identify the first candidate as a potential FSM. Candidates 2-4 are the same ones discussed in the previous section.

Listing 4: Excerpt of the topological analysis report of UART IP core obfuscated with *Active Hardware Metering*. Gate and net names have been blinded so that no information can be inferred by names. FFs $U1$, $U2$, $\ldots U9$ connect to control signals $O1$, $O2$, $O3$, $O4$. Boolean functions of the control signals are determined by Quine McCluskey Boolean function minimization, `~` refers to a negated literal, `+` to a disjunction, and `*` to a conjunction. Based on mutual configuration of FFs for all control signals, we identify FFs $U3$, …, $U9$ as obfuscation state memory since control signals only change their output in case these FFs hold a pre-defined all-zero value and consequently FFs $U1$ and $U2$ are marked as original state memory since they yield changing output for the control signals.

```
...

[Control Signal] O1 =  U1*           *~U5*~U6* U7*~U8* U9
[Control Signal] O2 = ~U1*           *~U5*~U6* U7*~U8*~U9
[Control Signal] O3 =  U1*           *~U5*~U6*~U7* U8*~U9
[Control Signal] O4 =     ~U2*~U3* U4* U5*~U6*~U7*~U8*~U9
...
```

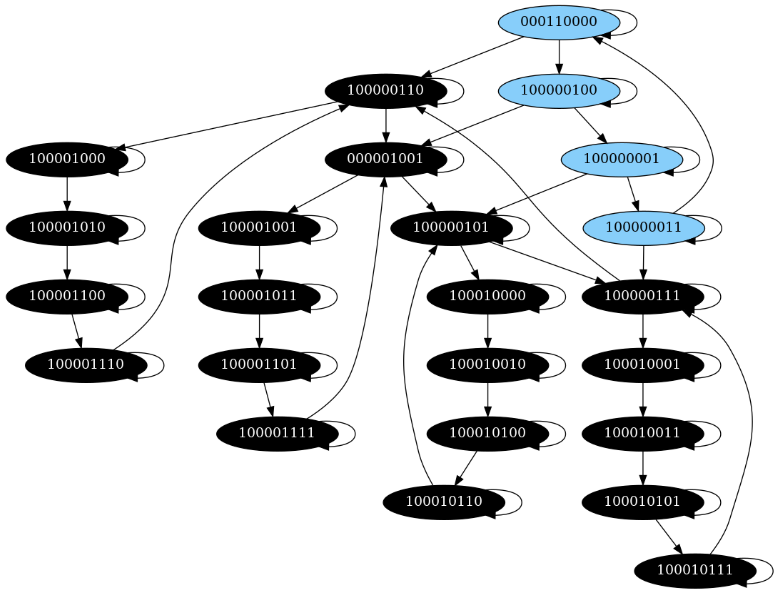


Figure 11: State transition graph of UART IP core obfuscated with *Active Hardware Metering*. Tarjan's algorithm splits all states in 2 strongly connected components: black hole cluster states (marked in black), and original FSM states (marked in blue). Input values for each state transition are deliberately left out for readability.

Listing 4 shows an excerpt of the topological analysis report of the selected FSM candidate. We see the minimized Boolean functions of control signals $O1$, $\ldots O4$ and can infer 4 potential states $s_1 = U1 \dots U9 = 100000101$, $s_2 = U1 \dots U9 = 100000100$, $s_3 = U1 \dots U9 = 100000010$, and $s_4 = U1 \dots U9 = 000110000$ which we feed to Boolean function analysis yielding the state transition graph in Figure 11. We want to highlight that the state values in Figure 11 are not in order $U1 \dots U9$ and thus differ from $s_1, \dots, s_4$. The computation time for the Boolean function analysis took around 2 s on a standard laptop. Using Tarjan's algorithm we were able to distinguish original states from black hole cluster states by virtue of the specific construction of *Active Hardware Metering*.

As explained in Section 4.4, the attack on the *Interlocking Obfuscation* scheme is the same for *Active Hardware Metering* and thus we deliberately omit the results for the former approach.

In summary, we observed that (semi-)automatic reverse engineering and targeted manipulation on FSM obfuscation performs well in practice for several obfuscation schemes and hardware designs. We also performed experiments where we integrated a UART communication interface with the cryptographic cores. Results for these experiments were similar to the ones presented in Section 5.1 and Section 5.2 so they have been omitted.

6 Hardware Nanomites

As demonstrated in previous sections, topological and Boolean function analyses are powerful tools that can extract crucial information from obfuscated FSM circuits. Even though the state transition graph may not be recovered by Boolean function analysis in some cases, topological analysis can be used to yield sufficient information to deobfuscate a protected FSM (e.g., for *Active Hardware Metering*).

Analogy: Software Obfuscation. In the domain of software protection, *self-modifying code*, i.e. code that alters its own instructions during execution, is typically leveraged to mitigate static analyses [41, 42, 43]. To additionally protect against dynamic analysis by use of debugging or emulation, alternative methods can be utilized [44]. A well-known example is *Armadillo Nanomites Technology* which protects software from dynamic analysis by exploiting the fact that only one ring-3 (user land) debugger can be attached to a debugged process at a time [45].

Nanomites in Hardware. Our idea for *Hardware Nanomites* builds on the concept of combining *self-modifying code* and *anti-debugging* to hinder reverse engineering of FSMs. From a high-level point of view, the state transition function and output logic of an FSM is split into partial designs that are loaded into an FPGA as needed via partial reconfiguration. This reconfiguration approach effectively provides the same design functionality as the original design at runtime without storing all logic needed to perform transitions in the FPGA at once. Intuitively, we hinder topological analysis since the entire state transition function is not present in the device at any point in time. The approach also hinders simulation, i.e. dynamic analysis, since currently available simulators cannot straightforwardly deal with partial dynamic FPGA reconfiguration. Thus, attempting to simulate a complete netlist representation of an FPGA-based FSM becomes an extremely challenging task.

6.1 Design

Principle. We now present the detailed design and implementation of *Hardware Nanomites* to hamper FSM reverse engineering, see Figure 12. In general, we unravel the overall FSM to split the state transition computation and output logic computation into multiple partial designs that are dynamically reconfigured to the *dynamic physical block* during runtime. Although one partial design per state is used in the following description for clarity, in reality, a larger number of states (e.g., 10-20) per partial design can be used for efficiency. Our formulation stated below can be easily modified to consider multiple states per partial design.

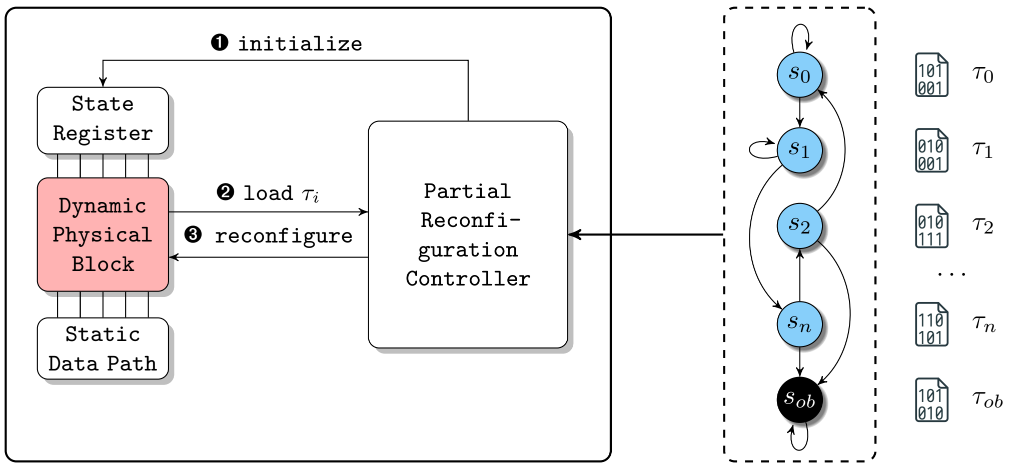


Figure 12: *Hardware Nanomites* system overview. Before using the first partial design τ_0, the state register is initialized to load it. After obtaining a request to load the next partial design τ_i, the partial reconfiguration controller then loads the requested design and reconfigures the dynamic physical block. The partial design interacts with the static data path and state register to perform calculations and afterwards requests loading of the next design. Note that state s_{ob} (marked in black) represents a bogus state.

Formally expressed, we form a tuple $\tau_i = (\lambda_i : \mathcal{S}_i \to \mathcal{O}_i, \delta_i : \mathcal{S} \times \mathcal{I} \to \mathcal{S}), \forall i = 0, \ldots, |\mathcal{S}| - 1$ and each τ_i is then implemented in a partial design. In addition to the original output and next state computation, we define bogus output values and bogus next state values for originally undefined or abnormal input configurations (e.g., so that a valid state may transit to one or multiple bogus states, see state s_{ob} in Figure 12), and bogus states transit between each other similar to the blackhole approach of *Dynamic State Deflection*. Bogus states are able to update the state register (to an apriori known invalid value) so that the next transition again yields a bogus state. Note that we map multiple state IDs to the same partial bitstream. When needed, each partial design is loaded into the reconfigurable partition connected to the static state register. After the processing of each state, the next state is stored in the state register and the *partial reconfiguration controller* is triggered to load the next partial design, if necessary. We note that *Hardware Nanomites* can be used in concert with an adapted key-based activation mechanism, i.e. a preceding obfuscation mode similar to *HARPOON* or *Active Hardware Metering*.

Example. To illustrate our approach, we consider an iterative AES cryptographic IP core consisting of five datapath layers *SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey*, and a round counter, see Figure 13. A single state per partial design is considered for illustrative purposes. The current state s_1 and currently loaded partial design determine the next state s_2 and control signals. Additionally, the computation conditionally triggers reconfiguration and also defines which partial design is reconfigured. Although no single partial design provides the complete functionality of the control path, the correct state transitions still take place since the entire state transition and output logic information is distributed across the partial designs.

Optimizations. To address performance penalties introduced by partial design reconfiguration (e.g., FPGA reconfiguration and the loading of partial designs), several techniques can be leveraged to improve performance: (1) *the use of multiple dynamic physical blocks*, and (2) *the use of partial bitstream caches.*

Since a dynamic physical block for a typical state transition function only consists of several hundred LUTs and FFs, see Section 6.2, we can easily instantiate multiple blocks to allow parallel pre-fetching of candidate partial designs. The selection of which partial

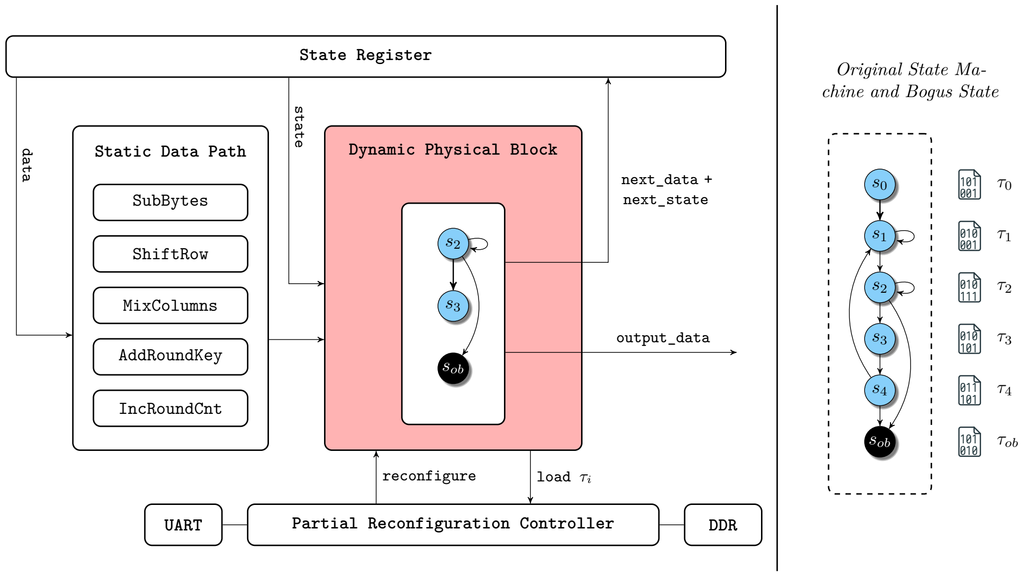


Figure 13: Example: Iterative AES IP core with *Hardware Nanomites* currently holding the partial design τ_2. The dynamic physical block (marked in red) takes the current state register value `state` and data from the static design part as an input and determines the next state and control signals. The circuit updates the round counter, assigns values to the primary output, and triggers partial reconfiguration, if necessary. Note that the state register is managed by the partial reconfiguration controller as a memory-mapped address.

design is used for a specific next state can then be determined by the partial reconfiguration controller. A series of multiplexers are needed at the state register input to select the correct next state in this scenario.

Another optimization strategy is the use of partial configuration generation [46] and caching [47] which have been investigated for years in the FPGA research community and are quickly becoming more widely used in practice. The use of partial reconfiguration to support partial designs with a selection of next state logic is particularly feasible considering the small footprint of most next state logic (typically less than ten LUTs per state transition). Thus, the amount of information that must be swapped into the FPGA on demand is highly constrained. In the caching scenario, the loading of partial bitstreams could be performed before next state processing takes place to reduce time overhead. Simultaneously, partial designs could be compressed to save space and loading time.

6.2 Implementation

We now highlight the details of our *Hardware Nanomites* implementation. Prior to the creation of the partial designs, the states of the FSM must be partitioned into groups. Each group defines a collection of state transition and output logic. This action is simple to define if the number of states per partial design is 1 or another small number. An initial approach for assignment of states to groups is to perform greedy state selection, although more complicated selection heuristics could be formulated for larger numbers of states per partition.

Our implementation of *Hardware Nanomites* incorporates several hardware components: (1) Xilinx HWICAP [48] to perform partial FPGA reconfiguration, and (2) a Xilinx

Microblaze soft-processor (performance optimized for standalone applications) [49] to simplify management of the partial bitstreams. The 64-bit state register is managed by the partial reconfiguration controller, i.e. as a memory-mapped address. Since bogus states are able to update the state register to invalid values, $2^{64} - (x \cdot m)$ bogus state values are reachable. Here x is the number of original states and m is the number of states that are additionally mapped to each original state. A 128-bit bus is used for the FSM input signals in our implementation which prevents a practical implementation of a Boolean function analysis. Also, we incorporated a Double Data Rate (DDR) controller to be able to store the partial bitstreams in external DRAM and allow for fast interfacing to the HWICAP. Since we store partial bitstreams externally, we can apply a randomly chosen permutation to access 128-bit chunks of each bitstream. This randomization hinders straightforward detection and translation of the partial bitstream into a (partial) gate-level netlist. To this end, we store the permutation and bitstream memory boundaries in the firmware so an adversary not only has to perform gate-level netlist reverse engineering but also low-level embedded software reverse engineering, see Section 6.3 for a detailed security analysis. As a result, the security of *Hardware Nanomites* can be improved by applying established software obfuscation or Instruction Set Architecture (ISA) obfuscation techniques [50] for the partial reconfiguration controller, see Section 6.3.

During the floor-planning step, the designer must mark a region that will host the dynamic part *a.k.a.* Dynamic Physical Block (pBlock). The region must be able to accommodate the largest partial module that will be loaded. To determine the size of the largest partial module, the designer synthesizes each partial module and obtains a count of the required resources from the synthesis report to allocate a suitable pBlock that is large enough to fit all partial modules.

On-Demand Partial Bitstream Streaming. Our physically-implemented prototype includes a UART core to receive partial bitstreams and store them in DRAM at runtime to provide on-demand design reconfiguration. This setup not only accelerates the implementation, but it may be of further interest for Internet of Things (IoT) applications where obfuscated hardware designs may be directly streamed to the FPGA to increase the effort of reverse engineering (similar to advanced software protection via code streaming [51]).

We now consider the hardware area overhead of *Hardware Nanomites* FSM obfuscation of the AES core implemented on a Xilinx KCU105 evaluation board [52] containing a Xilinx UltraScale FPGA.

***Hardware Nanomites* on Non-SoC FPGAs.** Table 1 and Table 2 show the hardware area overhead for both the static and dynamic design parts. The static part of the implementation utilizes roughly 13% of available LUTs resources and 6% of all FFs. The dynamic part utilizes only a fraction of the resources ($> 0.1\%$ of both LUTs and FFs). Note that the utilized storage memory scales with the number of partial bitstreams.

Hardware Nanomites are particularly well-suited for modern FPGA designs that implement partial reconfiguration features [53]. All FPGAs introduced by the two major FPGA vendors (Xilinx and Intel) over the past five years contain these features. If a reconfiguration controller is available, the area resource overhead of reconfiguration control logic is negligible. In terms of latency, the additional reconfiguration time depends on the pBlock size and bitstream reconfiguration speed. These values are also small for typical FSMs (e.g., $\frac{352kB}{400MB/s} = 880\mu s$ for a $400MB/s$ HWICAP speed and a partial bitstream size of $352kB$). Our utilized hardware components, the Xilinx Microblaze, a DDR controller and an UART controller, are typical components in hardware designs and can be easily repurposed.

***Hardware Nanomites* on SoC FPGAs.** Our *Hardware Nanomites* scheme can also easily leverage existing hardware resources on SoC FPGAs to minimize the resource overhead for the static design part. For example, on Xilinx 7-series ZYNQ devices, the fixed-logic ARM Cortex A9 can be used in place of the MicroBlaze. Additionally, a DDR controller, UART, and PCAP reconfiguration interface (replacing the HWICAP) are already present on the processing system [54, 55] and, thus, the only additional logic resource overhead is proportional to the added states for both storage and pBlock size.

Table 1: Hardware design resource utilization of the *Hardware Nanomites* static design parts synthesized for a XCKU040-2FFVA1156E. Note that the complete static design also includes components for the bitstream streaming such as a UART IP core.

Component	**#LUTs (Logic)**	**#FFs**	**#LUTs (Memory)**
Microblaze	1553 (0.64 %)	1401 (0.29 %)	198 (0.18 %)
DDR Controller	15151 (6.25 %)	17520 (3.61 %)	1379 (1.22 %)
HWICAP	312 (0.13 %)	959 (0.20 %)	1 ($\geq$ 0.01 %)
AXI SmartConnect	5827 (2.40 %)	8977 (1.85 %)	2017 (1.79 %)
Misc. Parts (UART, ...)	1335 (0.55 %)	1752 (0.36 %)	94 (0.08 %)
Complete Static Design	24178 (9.97 %)	30609 (6.31 %)	3689 (3.27 %)

Table 2: Hardware design resource utilization for the pBlock (*Hardware Nanomites* dynamic design part) synthesized for a XCKU040-2FFVA1156E.

#LUTs (Logic)	**#FFs**	**#LUTs (Memory)**	**Partial Bitstream Size**
160 (0.07 %)	320 (0.07 %)	80 (0.07 %)	352 kByte

Case Study: AES. In our evaluation we used an adapted version of the AES-T1000 implementation [56] consisting of a 5-state FSM (similar to Figure 14 in the Appendix) that includes bogus states for transitions that were originally invalid. Moreover note that we changed original cipher architecture to an iterative one. Prior to loading a new functional partial bitstream of $352kB$ in an UltraScale device, a clearing bitstream of $9.9kB$ must be loaded. Note that the clearing bitstream is not needed for UltraScale+ devices. For a standard reconfiguration speed of $400MB/s$, the reconfiguration time overhead for all five states is $\frac{(352kB+9.9kB)\cdot 5}{400MB/s} = 4.524 \cdot 10^{-3}s$. Since our AES design clock frequency is limited to $100MHz$ (the speed of the DDR controller), the AES operation is delayed $4.524 \cdot 10^{-3}s \cdot 100 \cdot 10^{6}s^{-1} = 452,375$ clock cycles.

Since we believe that *Hardware Nanomites* provide a valuable building block for FPGA-based hardware obfuscation, we plan to publish the sources for our implementation to foster future research in this direction.

6.3 Security Evaluation

Before we present a security evaluation of *Hardware Nanomites*, we stress that sound quantification of resiliency against reverse engineering is an open problem since it involves the quantification of human factors [2]. Therefore, we can only present what steps a reverse engineer must overcome to analyze an FSM protected with *Hardware Nanomites*. To this end, we assess the security for static design analysis and dynamic design analysis, i.e. simulation, and we discuss why the mentioned approaches are challenging to reverse engineer in an automated fashion. Note that we leverage the same adversary model as defined in Section 2.1, i.e. access to the original obfuscated gate-level netlist and the adversary is able (as soon as he has access to the partial bitstreams) to convert them into partial netlists. The attacker's goal is to deobfuscate the design (e.g., identify activation keys or tamper with the design to skip key validation.)

6.3.1 Static Design Analysis

As shown in previous sections, static analysis techniques, such as topological analysis, are effective in obtaining crucial FSM information from gate-level netlists. Thus, from a high-level point of view, the adversary's goal is to transform the static design and all partial designs into a version where known FSM reverse engineering techniques can be repurposed. To this end, an adversary may attempt to multiplex the partial designs into a single netlist which could eventually be prone to topological analysis. However, for this generic reverse engineering activity, the adversary encounters two major challenges: (1) access to the partial bitstreams, and (2) mapping of the τ_i IDs to partial bitstreams.

Challenge: Access to Partial Bitstreams. To transform the partial designs into a single static gate-level netlist, the adversary has to first obtain the partial bitstreams. Depending on the implementation, partial bitstreams are either stored in FPGA internal or external memory. In the former case, the adversary must identify and reverse engineer the contents of FPGA memory structures that hold the partial bitstreams. There are no automated techniques reported so far to the best of our knowledge that perform this activity. In the latter case (which we use in our implementation), the adversary must first analyze the chunk-level permutations of partial bitstreams which requires an analysis of the partial reconfiguration controller ($\approx 1.7k$ LUTs and $\approx 1.4k$ FFs) as there are numerous variants for the Xilinx Microblaze processor and significant parts of the firmware data flow ($\approx 4.4MB$), i.e. combined hardware and software reverse engineering. These numbers for the effort depict the worst case of reverse engineering all information. Since access to the partial bitstreams involves combined hardware and software reverse engineering, it is quite challenging considering state-of-the-art reverse engineering technology.

Challenge: Mapping τ_i IDs to Partial Bitstream. Once the adversary has obtained the partial bitstreams and transformed them into partial netlists (via file format reverse engineering), he must assemble a single static netlist. To this end, all partial netlists must be multiplexed to determine which state transition is currently active. Note that this action requires an understanding of the mapping between τ_i indices and the partial bitstreams. This mapping is handled in the firmware of the reconfiguration controller and thus extracting this information requires a combination of hardware and software reverse engineering efforts. After successful extraction of this mapping, the adversary is able to assemble the netlist and finally perform topological analysis.

In summary, for an attacker to defeat our approach substantial new automated software and hardware reverse engineering approaches that have not yet been reported would be needed. In contrast to the aforementioned state-of-the-art FSM obfuscation schemes, we significantly increase the required manual reverse engineering effort since at least several thousand logic gates and several kilobytes of firmware must be analyzed rather than several hundred FSM gates and signals. Although there is a large body of software reverse engineering research, such automated software analyses can be easily thwarted by use of software and ISA obfuscation [50] so that manual deobfuscation of the ISA, software, and hardware must be performed first.

6.3.2 Dynamic Design Analysis

Another generic strategy to obtain operational information from an FSM equipped with *Hardware Nanomites* is to analyze runtime behavior, i.e. perform simulation. However, to the best of our knowledge, an efficient gate-level simulation model for partially reconfigurable FPGA designs is not currently available in industry or academia. While rudimentary approaches exist [57, 58], they either tend to drastically increase the size of the simulation model and thus increase the workload for the simulator beyond practical limits, or they are not designed for gate-level simulation. We acknowledge that designing and implementing such a tool (or improving the rudimentary ones) is theoretically possible and would allow inspection of the dynamic physical block during runtime. Even with such a tool, reverse engineering of a *Hardware Nanomites* protected FSM would be a challenging process as both software and hardware have to be analyzed in concert since the reconfiguration controller is implemented in software.

Another possible dynamic reverse engineering strategy would be to derive a static simulation model for each partial design that includes the static part, so that the reverse engineer is able to iteratively analyze and switch between different simulation models manually. However, this approach requires significant apriori (manual) static reverse engineering effort, i.e. identification of partial bitstreams and extraction of ID to bitstream mappings. As mentioned in the previous section, no semi-automated reverse engineering techniques that perform these tasks have been reported to the best of our knowledge.

7 Discussion

Applications of FSM Reverse Engineering. We have demonstrated that automated reverse engineering of FSMs provides valuable high-level information, i.e. design of the control path and its controlled datapath units. Since reverse engineering is a tool to enable legitimate and illegitimate applications, we discuss the implications for both.

From a defender's point of view, high-level FSM information can be used to identify hardware Trojans implemented with sequential logic or Trojans which incorporate FSM-based obfuscation for increased stealthiness [30, 56, 59]. Moreover, our insights on the capabilities of FSM reverse engineering can support assessment of future hardware design obfuscation strategies. Reverse engineering is beneficial in the case of source code loss, faulty product design detection, and competitor analysis [3].

From an adversary's point of view, high-level FSM information offers an attractive entry point for further reverse engineering of datapath units such as details of a cryptographic implementation or microarchitecture specifics. More importantly, register grouping discloses crucial module boundary information, which partitions the design into easier to analyze, functionally-related units.

On the Difficulty of Using Obfuscation Metrics. Modeling the security of practical obfuscation schemes against reverse engineering is challenging, see Section 4. We observed that a realistic appreciation of reverse engineering capabilities and consideration of the system context are often neglected. The capabilities of (automated) gate-level netlist reverse engineering are often not considered, since this topic is not well studied, see Section 2.2. Designers of obfuscation techniques often do not detail why the development of automated reverse engineering is challenging or what steps would be needed by a rational reverse engineer to defeat the obfuscation. In Section 4.5 we provided a concise overview of the lessons learned to provide future obfuscation designers with valuable information on adversarial approaches.

Focusing solely on FSM obfuscation is not enough to prevent an attacker from gaining meaningful high-level design information. The general system context must be considered as well. Consider an FPGA hardware design with a communication interface, a cryptographic algorithm, and an FSM controlling both cores. The FSM is obfuscated and employs a key-based activation, so that it is not possible to algorithmically or manually reverse engineer its state transition function (e.g., 64-bit input signal). This implementation enables the adversary to obtain high-level design information without analyzing the *highly-obfuscated* FSM. This example shows that it is necessary to consider obfuscation not *just* for the FSM, but also for the system as a whole. This directive extends to people researching obfuscation as well as system designers responsible for implementing obfuscation techniques. Designers need to consider the non-obfuscated parts of a design, even in the presence of FSM obfuscation.

As mentioned in Section 5, we acknowledge that we implemented the different FSM obfuscation schemes ourselves as no publicly available implementations were available. However, our tools did not use any design information beyond what was obtained from the gate-level netlists.

Future Work Several directions can be explored in future research. Reverse engineering of FPGA designs with dynamic reconfiguration should be explored to quantify the complexity increase (compared to static designs). In addition, further work should explore automated techniques for general-purpose reverse engineering for security-relevant circuitry. It would be desirable to quantify the human factor in reverse engineering or to set up a reverse engineering competition. The evaluation can be extended to diverse (open-source and closed-source) synthesizers to potentially improve reliability of the topological analysis.

8 Conclusion

Due to globalization, IC design, implementation, and manufacturing involves various (untrusted) suppliers and stakeholders. Hence, a designer's valuable IP is visible to many parties thus increasing the risk of IP piracy. Numerous works have proposed solutions to protect against IP piracy. Hardware design obfuscation of a circuit's FSM to protect against the crucial threat of reverse engineering and IP infringement has been a particular focus. However, the security of many schemes is in doubt since realistic reverse engineering capabilities are barely addressed in the open literature and thus not adequately considered in security analyses.

In this paper, we carefully reviewed the security of several state-of-the-art FSM obfuscation schemes. In concert with realistic reverse engineering capabilities, we demonstrated several generic strategies to bypass these schemes on FPGA gate-level netlists while keeping analysis times practical. We augment netlist reverse engineering algorithms to disclose high-level FSM information in FPGA gate-level netlists. Our rigorous evaluation demonstrates the effectiveness of FSM reverse engineering and the automatically disclosed information supports a human analyst to further reverse engineer a design for constructive as well as malicious purposes.

Our insights on realistic reverse engineering capabilities invite a rethinking of future hardware design obfuscation.

Acknowledgments

The authors would like to thank our anonymous reviewers and our shepherd Colin O'Flynn for their valuable feedback and remarks. Part of this work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation programme (ERC Advanced Grant No. 695022 (EPoCH)) and by NSF grant CNS-1563829.

References

[1] M. Fyrbiak, S. Wallat, P. Swierczynski, M. Hoffmann, S. Hoppach, M. Wilhelm, T. Weidlich, R. Tessier, and C. Paar, "HAL—the missing piece of the puzzle for hardware reverse engineering, trojan detection and insertion," IEEE Transactions on Dependable and Secure Computing, 2018, to appear.

[2] M. Fyrbiak, S. Strauss, C. Kison, S. Wallat, M. Elson, N. Rummel, and C. Paar, "Hardware reverse engineering: Overview and open challenges," in IEEE 2nd International Verification and Security Workshop, IVSW 2017,

Thessaloniki, Greece, July 3-5, 2017. IEEE, 2017, pp. 88–94. [Online]. Available: https://doi.org/10.1109/IVSW.2017.8031550

[3] A. Vijayakumar, V. C. Patil, D. E. Holcomb, C. Paar, and S. Kundu, "Physical design obfuscation of hardware: A comprehensive investigation of device and logic-level techniques," IEEE Trans. Information Forensics and Security, vol. 12, no. 1, pp. 64–77, 2017. [Online]. Available: https://doi.org/10.1109/TIFS.2016.2601067

[4] B. Shakya, M. M. Tehranipoor, S. Bhunia, and D. Forte, Introduction to Hardware Obfuscation: Motivation, Methods and Evaluation, 1st ed. Springer Publishing Company, Incorporated, 2017, vol. 1, pp. 3–32.

[5] R. S. Chakraborty and S. Bhunia, "Hardware protection and authentication through netlist level obfuscation," in ICCAD. IEEE Computer Society, 2008, pp. 674–677.

[6] ——, "Security against hardware trojan through a novel application of design obfuscation," in 2009 International Conference on Computer-Aided Design, ICCAD 2009, San Jose, CA, USA, November 2-5, 2009, J. S. Roychowdhury, Ed. ACM, 2009, pp. 113–116. [Online]. Available: http://doi.acm.org/10.1145/1687399.1687424

[7] ——, "Security through obscurity: An approach for protecting register transfer level hardware IP," in IEEE International Workshop on Hardware-Oriented Security and Trust, HOST 2009, San Francisco, CA, USA, July 27, 2009. Proceedings, M. Tehranipoor and J. Plusquellic, Eds. IEEE Computer Society, 2009, pp. 96–99. [Online]. Available: https://doi.org/10.1109/HST.2009.5224963

[8] ——, "HARPOON: an obfuscation-based soc design methodology for hardware protection," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 28, no. 10, pp. 1493–1502, 2009. [Online]. Available: https://doi.org/10.1109/TCAD.2009.2028166

[9] Y. Alkabani and F. Koushanfar, "Active hardware metering for intellectual property protection and security," in USENIX Security Symposium. USENIX Association, 2007.

[10] F. Koushanfar, "Provably secure active IC metering techniques for piracy avoidance and digital rights management," IEEE Trans. Information Forensics and Security, vol. 7, no. 1, pp. 51–63, 2012.

[11] S. Amir, B. Shakya, D. Forte, M. Tehranipoor, and S. Bhunia, "Comparative analysis of hardware obfuscation for IP protection," in ACM Great Lakes Symposium on VLSI. ACM, 2017, pp. 363–368.

[12] S. E. Quadir, J. Chen, D. Forte, N. Asadizanjani, S. Shahbazmohamadi, L. Wang, J. A. Chandy, and M. Tehranipoor, "A survey on chip to system reverse engineering," JETC, vol. 13, no. 1, pp. 6:1–6:34, 2016. [Online]. Available: http://doi.acm.org/10.1145/2755563

[13] J. Note and É. Rannaud, "From the bitstream to the netlist," in FPGA. ACM, 2008, p. 264.

[14] P. Swierczynski, M. Fyrbiak, P. Koppe, A. Moradi, and C. Paar, "Interdiction in practice - hardware trojan against a high-security USB flash drive," J. Cryptographic Engineering, vol. 7, no. 3, pp. 199–211, 2017.

[15] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," Proceedings of the IEEE, vol. 102, no. 8, pp. 1283–1295, 2014. [Online]. Available: https://doi.org/10.1109/JPROC.2014.2335155

[16] M. C. Hansen, H. Yalcin, and J. P. Hayes, “Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering,” IEEE Design & Test of Computers, vol. 16, no. 3, pp. 72–80, 1999.

[17] Y. Shi, C. W. Ting, B. Gwee, and Y. 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. IEEE, 2010, pp. 2610–2613. [Online]. Available: https://doi.org/10.1109/ISCAS.2010.5537093

[18] Y. Shi, B. Gwee, Y. Ren, T. K. Phone, and C. W. Ting, “Extracting functional modules from flattened gate-level netlist,” in ISCIT. IEEE, 2012, pp. 538–543.

[19] W. Li, Z. Wasson, and S. A. Seshia, “Reverse engineering circuits using behavioral pattern mining,” in HOST. IEEE, 2012, pp. 83–88.

[20] W. Li, A. Gascón, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. 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. IEEE Computer Society, 2013, pp. 67–74. [Online]. Available: https://doi.org/10.1109/HST.2013.6581568

[21] P. Subramanyan, N. Tsiskaridze, W. Li, A. Gascón, W. Y. Tan, A. Tiwari, N. Shankar, S. A. Seshia, and S. Malik, “Reverse engineering digital circuits using structural and functional analyses,” IEEE Trans. Emerging Topics Comput., vol. 2, no. 1, pp. 63–80, 2014. [Online]. Available: https://doi.org/10.1109/TETC.2013.2294918

[22] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanovic, and S. Malik, “Template-based circuit understanding,” in FMCAD. IEEE, 2014, pp. 83–90.

[23] T. Meade, S. Zhang, and Y. Jin, “Netlist reverse engineering for high-level functionality reconstruction,” in ASP-DAC. IEEE, 2016, pp. 655–660.

[24] S. Wallat, M. Fyrbiak, M. Schlogel, and C. 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. IEEE, 2017, pp. 95–100. [Online]. Available: https://doi.org/10.1109/IVSW.2017.8031551

[25] T. Meade, Z. Zhao, S. Zhang, D. Z. Pan, and Y. Jin, “Revisit sequential logic obfuscation: Attacks and defenses,” in ISCAS. IEEE, 2017, pp. 1–4.

[26] R. A. Saleh, S. J. E. Wilton, S. Mirabbasi, A. J. Hu, M. R. Greenstreet, G. Lemieux, P. P. Pande, C. Grecu, and A. Ivanov, “System-on-chip: Reuse and integration,” Proceedings of the IEEE, vol. 94, no. 6, pp. 1050–1069, 2006. [Online]. Available: https://doi.org/10.1109/JPROC.2006.873611

[27] F. Koushanfar, Hardware Metering: A Survey. Springer Publishing Company, Incorporated, 2012, pp. 103–122.

[28] R. E. Tarjan, “Depth-first search and linear graph algorithms,” SIAM J. Comput., vol. 1, no. 2, pp. 146–160, 1972.

[29] W. V. Quine, “The problem of simplifying truth functions,” The American Mathematical Monthly, vol. 59, no. 8, pp. 521–531, 1952.

[30] T. Meade, Y. Jin, M. Tehranipoor, and S. Zhang, “Gate-level netlist reverse engineering for hardware security: Control logic register identification,” in ISCAS. IEEE, 2016, pp. 1334–1337.

[31] Spartan-6 Libraries Guide for HDL Designs, Xilinx, 2009, v 11.4.

[32] J. Dofe and Q. Yu, “Novel dynamic state-deflection method for gate-level design obfuscation,” IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 37, no. 2, pp. 273–285, 2018.

[33] J. Dofe, Y. Zhang, and Q. Yu, “DSD: A dynamic state-deflection method for gate-level netlist obfuscation,” in ISVLSI. IEEE Computer Society, 2016, pp. 565–570.

[34] F. Koushanfar and G. Qu, “Hardware metering,” in DAC. ACM, 2001, pp. 490–493.

[35] F. Koushanfar, Active Hardware Metering by Finite State Machine Obfuscation, 1st ed. Springer Publishing Company, Incorporated, 2017, vol. 1, pp. 161–187.

[36] S. Gören, O. Ozkurt, A. Yildiz, H. F. Ugurdag, R. S. Chakraborty, and D. Mukhopadhyay, “Partial bitstream protection for low-cost fpgas with physical unclonable function, obfuscation, and dynamic partial self reconfiguration,” Computers & Electrical Engineering, vol. 39, no. 2, pp. 386–397, 2013.

[37] A. R. Desai, M. S. Hsiao, C. Wang, L. Nazhandali, and S. Hall, “Interlocking obfuscation for anti-tamper hardware,” in CSIIRW. ACM, 2013, p. 8.

[38] T. Kerins and K. Kursawe, “A cautionary note on weak implementations of block ciphers,” in In 1st Benelux Workshop on Information and System Security (WISSec 2006), 2006, p. 12.

[39] P. Swierczynski, M. Fyrbiak, P. Koppe, and C. Paar, “FPGA trojans through detecting and weakening of cryptographic primitives,” IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 34, no. 8, pp. 1236–1249, 2015. [Online]. Available: https://doi.org/10.1109/TCAD.2015.2399455

[40] S. Trimberger and J. Moore, “FPGA security: Motivations, features, and applications,” Proceedings of the IEEE, vol. 102, no. 8, pp. 1248–1265, 2014. [Online]. Available: https://doi.org/10.1109/JPROC.2014.2331672

[41] D. Aucsmith, “Tamper resistant software: an implementation,” in Information Hiding, R. Anderson, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 317–333.

[42] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto, “Exploiting self-modification mechanism for program protection,” in Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003, Nov 2003, pp. 170–179.

[43] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere, “Software protection through dynamic code mutation,” in Information Security Applications, J.-S. Song, T. Kwon, and M. Yung, Eds. Springer Berlin Heidelberg, 2006, pp. 194–206.

[44] M. N. Gagnon, S. Taylor, and A. K. Ghosh, “Software protection through anti-debugging,” IEEE Security Privacy, vol. 5, no. 3, pp. 82–84, May 2007.

[45] A. Inc., “Nanomite and debug blocker technologies: Scheme, pros, and cons,” https://www.apriorit.com/white-papers/293-nanomite-technology.

[46] R. Tessier, K. Pocek, and A. DeHon, “Reconfigurable computing architectures,” Proceedings of the IEEE, vol. 103, no. 3, pp. 332–354, March 2015.

[47] Z. Li, K. Compton, and S. Hauck, "Configuration caching management techniques for reconfigurable computing," in Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No.PR00871), 2000, pp. 22–36.

[48] Xilinx, AXI HWICAP v3.0 LogiCORE IP Product Guide, 2016, https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf.

[49] ——, "Microblaze soft processor core," https://www.xilinx.com/products/design-tools/microblaze.html.

[50] M. Fyrbiak, S. Rokicki, N. Bissantz, R. Tessier, and C. Paar, "Hybrid obfuscation to protect against disclosure attacks on embedded microprocessors," IEEE Transactions on Computers, vol. 67, no. 3, pp. 307–321, March 2018.

[51] I. Artzi, B. Mcdermott, D. Eylon, A. Ramon, and Y. Volk, "Network streaming of multi-application program code," May 2006. [Online]. Available: http://www.freepatentsonline.com/7051315.html

[52] Xilinx, KCU105 Board User Guide, 2017, https://www.xilinx.com/support/documentation/boards_and_kits/kcu105/ug917-kcu105-eval-bd.pdf.

[53] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Oct 2016, pp. 1–13.

[54] Xilinx, Zynq-7000 All Programmable SoC Data Sheet: Overview, 2017, https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[55] ——, Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite for Zynq-7000 AP SoC Processor, 2015, https://www.xilinx.com/support/documentation/application_notes/xapp1231-partial-reconfig-hw-accelerator-vivado.pdf.

[56] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in 2013 IEEE 31st International Conference on Computer Design, ICCD 2013, Asheville, NC, USA, October 6-9, 2013. IEEE Computer Society, 2013, pp. 471–474. [Online]. Available: https://doi.org/10.1109/ICCD.2013.6657085

[57] J. Stockwood and P. Lysaght, "A simulation tool for dynamically reconfigurable field programmable gate arrays," in Proceedings of Eighth International Application Specific Integrated Circuits Conference, Sep 1995, pp. 167–170.

[58] X. Peña, F. Rincon, J. Dondo, J. Caba, and J. C. Lopez, "Run-time partial reconfiguration simulation framework based on dynamically loadable components," in Applied Reconfigurable Computing, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Springer International Publishing, 2015, pp. 153–164.

[59] J. Zhang, F. Yuan, and Q. Xu, "Detrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in ACM Conference on Computer and Communications Security. ACM, 2014, pp. 153–166.

[60] N. S. Agency, "Rijndael128 implementation," http://csrc.nist.gov/archive/aes/round2/r2anlsys.htm#NSA, 2000, [Online] Accessed: July 24, 2018.

[61] IEEE 2nd International Verification and Security Workshop, IVSW 2017, Thessaloniki, Greece, July 3-5, 2017. IEEE, 2017. [Online]. Available: http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=8024489

[62] D. Forte, S. Bhunia, and M. M. Tehranipoor, Eds., Hardware Protection Through Obfuscation, 1st ed. Springer Publishing Company, Incorporated, 2017, vol. 1.

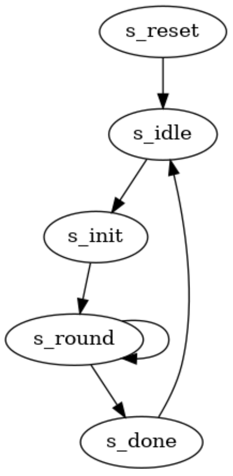
Appendix

Listing 5 shows an FSM with a merged counter (highlighted lines in yellow). Note the FSM is implemented in an iterative AES core to control key schedule processing. Since the round counter signal `CV_RUNUP_STATE` is processed inside the control path, the counter and the state signal `RUNUP_STATE` are merged into a single FSM candidate. Our *input independent state series* analysis (presented in Section 3.2) addresses this issue and splits the counter from the FSM part.

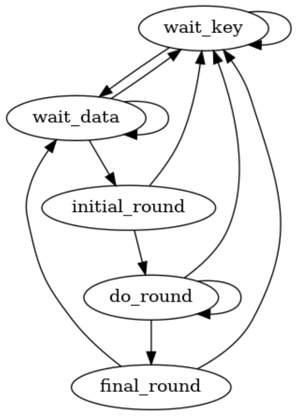
Listing 5: FSM incorporating a counter (see `key_sched_iterative.vhdl` [60]).

```
...
constant LAST_ECVRUNUP_STEP : integer := 1; -- # of steps for cv runup
constant LAST_DCVRUNUP_128 : integer := 9; -- # of steps for cv runup

signal CV_RUNUP_STEP : integer range 0 to 255;
type RUNUP_STATE_TYPE is (HOLD, CV_RUNUP, CV_EXPAND, DONE);
signal RUNUP_STATE : RUNUP_STATE_TYPE;

...

RUNUP_FLOW: process(clock, reset)
begin
  if reset = '1' then
    CV_RUNUP_STEP <= 0;
    RUNUP_STATE  <= HOLD;
  elsif clock'event and clock = '1' then
    case RUNUP_STATE is

    ...

      when CV_RUNUP =>
        if ( CV_RUNUP_STEP /= LAST_ECVRUNUP_STEP and KS_ENC = '1')
        or ( CV_RUNUP_STEP /= LAST_DCVRUNUP_128 and KS_ENC = '0' ) then
            CV_RUNUP_STEP <= CV_RUNUP_STEP + 1;
            RUNUP_STATE  <= RUNUP_STATE;
        else
            RUNUP_STATE  <= DONE;
            CV_RUNUP_STEP <= 0;
        end if;

        ...

      end case;
   end if; -- reset = '1'
end process; -- RUNUP_FLOW
```

Figure 14 shows the reduced state transition graphs of the hardware designs utilized in our evaluation, see Section 5. We deliberately omitted the input signals yielding state transitions for improved readability. Note that we retained original state names for clarity as our Boolean function analysis only recovers the state memory value but obviously not the original meaning of the state.



(a) SHA-3 FSM.



(b) AES encryption FSM.

Figure 14: Original state transition graph diagrams of hardware designs utilized in Section 5.