Technische Universität München
TUM School of Computation, Information and Technology

New Perspectives on Sequential Reverse Engineering and Obfuscation

Michaela Franziska Brunner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung einer

Doktorin der Ingenieurwissenschaften (Dr.-Ing.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr.-Ing. Robert Wille

Prüfende der Dissertation:
1. Prof. Dr.-Ing. Georg Sigl
2. Priv.-Doz. Dr.-Ing. habil. Bing Li

Die Dissertation wurde am 31.01.2024 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 07.05.2024 angenommen.

To my husband.

Abstract

To cope with the high demand for cheap chips, nowadays, chips are predominantly not produced, tested, or packaged in the design house's own factories but in third-party factories. This raises the risk for the security of chips' intellectual property. Serious threats are, for example, malicious design modifications or the theft of the chips' intellectual property. Hardware gate-level netlist reverse engineering can support these attacks by increasing the understanding of a chip's design and functionality. On the other hand, it can also contribute to verifying the integrity of end products.

Of special interest to a reverse engineer is the control logic of a design. The control logic or finite state machine is usually the most costly and time-consuming part when designing a chip, as it is no standardized component but represents the unique functionality of each individual design. Thus, it is an interesting target for competing companies to gain an advantage, for attackers to manipulate the design's functionality, or for chip owners to uncover potential malicious manipulations. Sequential hardware gate-level netlist reverse engineering reveals and analyzes sequential parts (state flip-flops) and combinational parts in a circuit that belong to the state machine, while state machine obfuscation protects state machines against sequential reverse engineering.

This thesis contributes to the increasingly important challenges of protecting the chips' intellectual property by taking a novel perspective on sequential reverse engineering and obfuscation. In particular, it benefits from the analysis of features in hardware gate-level netlists and from the combination of techniques.

The thesis derives a novel concept of how to categorize state flip-flop sets. For this, the work analyzes the degrees of freedom of the well-known state machine descriptions and develops state flip-flop and state flip-flop set properties based on these analysis results. Additionally, the work presents a definition for a special state flip-flop set: the state flip-flop set definition for a human-readable state machine. This definition uses the newly derived state flip-flop (set) properties and defines what flip-flops are assumed to belong to a human-readable state machine, i.e. to the original state machine without tons of extra information.

In addition to the feature analysis of state flip-flops, the thesis also analyzes the features of methods that identify state flip-flops. The work concludes what features are prevalent for identification methods and what features could be exploited to hinder identification methods.

The thesis develops four post-processing techniques to improve the results of state flip-flop identification methods. The post-processing methods use the derived state flip-flop properties to add flip-flops to and/or remove flip-flops from potential state flip-flop sets, which were output by state flip-flop identification methods. The evaluation shows that post-processing largely improves the original results of state flip-flop identification methods which are mainly based on similarity-based features.

In addition to improving sequential reverse engineering, the thesis also develops two novel state machine obfuscation techniques. The first technique applies a camouflaging method, Timing Camouflage, on state flip-flops. Timing Camouflage removes flip-flops while preserving the original functionality by adjusting the circuit timing properties accordingly. The removed state flip-flops can no longer be identified by sequential gate-level netlist reverse engineering,

thus hindering successful sequential reverse engineering. As Timing Camouflage can only be applied on flip-flops with no combinational feedback path, the work additionally develops two state machine redesign techniques to avoid combinational feedback paths for the state flip-flop which Timing Camouflage should remove. The second obfuscation technique uses the derived common and exploitable features of state flip-flop identification methods. It adds a—for the state flip-flop identification method highly attractive—honeypot state machine and translates the original state machine into a—for the state flip-flop identification method highly unattractive—state machine without changing its actual functionality. As a result, a state flip-flop identification technique identifies the attractive honeypot state machine and not the unattractive original state machine as the best state machine candidate or can no longer identify the correct original state machine. This complicates or hinders a successful sequential reverse engineering.

Lastly, the thesis also concentrates on possible negative consequences that protection mechanisms can cause. It presents a logic locking induced fault attack that uses the protection technique called logic locking, to insert faults into a gate-level netlist. Logic locking adds key-controlled locking gates into a netlist. It ensures that the circuit only functions correctly if the correct secret locking key is applied. However, modifying a locking key bit corresponds to a fault injection. The work analyzes the applicability of this inserted fault based on the used locking technique, key management, and fault analysis technique. Finally, a use case demonstrates the logic locking induced fault attack by extracting a secret cryptographic key.

Zusammenfassung

Um die hohe Nachfrage an billigen Chips zu bewältigen, werden diese heutzutage überwiegend in Fabriken Dritter und nicht in den eigenen Fabriken des Designers hergestellt, getestet oder verpackt. Dadurch erhöht sich das Risiko für die Sicherheit des geistigen Eigentums von Chips. Ernsthafte Bedrohungen sind beispielsweise bösartige Änderungen am Design oder der Diebstahl des geistigen Eigentums von Chips. Das Hardware Reverse Engineering von Netzlisten auf Gatterebene kann solche Angriffe unterstützen, indem es das Verständnis für das Design und das Verständnis über die Funktionalität eines Chips verbessert. Auf der anderen Seite kann es auch dazu beitragen, die Integrität von Endprodukten zu überprüfen.

Von besonderem Interesse für einen Reverse Engineer ist die Kontrolllogik eines Designs. Die Kontrolllogik bzw. der endliche Zustandsautomat ist keine standardisierte Komponente, sondern sie bzw. er beschreibt die einzigartige Funktionalität jedes einzelnen Designs. Somit ist dies in der Regel der kostspieligste und zeitaufwändigste Teil bei der Entwicklung eines Chips. Das macht endliche Zustandsautomaten zu einem interessanten Ziel für konkurrierende Unternehmen, um sich einen Vorteil zu verschaffen, für Angreifer, um die Funktionalität des Designs zu manipulieren, oder für Chipeigentümer, um mögliche, böswillige Manipulationen aufzudecken. Sequentielles Hardware Reverse Engineering von Netzlisten auf Gatterebene identifiziert und analysiert sequenzielle Komponenten (Zustandsflipflops) und kombinatorische Komponenten einer Schaltung, die einem Zustandsautomaten angehören. Im Gegensatz dazu schützt die Zustandsautomaten-Obfuskation einen Zustandsautomaten vor sequenziellem Reverse Engineering.

Diese Arbeit leistet einen Beitrag zu den immer wichtiger werdenden Herausforderungen beim Schutz des geistigen Eigentums von Chips, indem sie eine neuartige Perspektive auf sequentielles Reverse Engineering und Obfuskation wirft. Dabei profitiert sie insbesondere von der Analyse von Merkmalen in Hardware Netzlisten auf Gatterebene und von der Kombination von Techniken.

Die Arbeit erstellt ein neuartiges Konzept zur Kategorisierung von Sets aus Zustandsflipflops. Dazu analysiert sie die Freiheitsgrade der bekannten Zustandsmaschinenbeschreibungen und ermittelt auf Basis dieser Analyseergebnisse Merkmale von Zustandsflipflops und Sets aus Zustandsflipflops. Darüber hinaus präsentiert die Arbeit eine Definition für ein spezielles Set aus Zustandsflipflops: die Definition für ein Set aus Zustandsflipflops für eine menschenlesbare Zustandsmaschine. Die Definition verwendet die neu hergeleiteten Merkmale eines Zustandsflipflops oder Sets aus Zustandsflipflops und definiert, bei welchen Flipflops davon ausgegangen wird, dass diese zu einem menschenlesbaren Zustandsautomaten, also dem ursprünglichen Zustandsautomaten ohne eine große Menge zusätzlicher Informationen, gehören.

Die Arbeit analysiert aber nicht nur die Merkmale von Zustandsflipflops, sondern auch die Merkmale von Methoden zur Identifizierung von Zustandsflipflops. Die Arbeit zieht Schlussfolgerungen darüber, welche Merkmale bei Identifizierungsmethoden häufig auftreten und welche Merkmale ausgenutzt werden könnten, um Identifizierungsmethoden zu erschweren.

Die Arbeit entwickelt vier Nachbearbeitungstechniken, um die Ergebnisse von Methoden, die Zustandsflipflops identifizieren, zu verbessern. Die Nachbearbeitungsverfahren verwenden die hergeleiteten Merkmale der Zustandsflipflops, um Flipflops zu potentiellen Sets aus

Zustandsflipflops, die durch Methoden zur Identifizierung von Zustandsflipflops ausgegeben wurden, hinzuzufügen und/oder von diesen zu entfernen. Die Auswertung zeigt, dass die Nachbearbeitung die ursprünglichen Ergebnisse einer Zustandsflipflopidentifizierung, die sich hauptsächlich auf ähnlichkeitsbasierte Merkmale stützt, überwiegend verbessert.

Neben der Verbesserung des sequentiellen Reverse Engineerings werden in dieser Arbeit auch zwei neuartige Obfuskationstechniken für Zustandsautomaten entwickelt. Die erste Technik wendet eine Camouflaging Methode, die als Timing Camouflage bezeichnet wird, auf Zustandsflipflops an. Timing Camouflage entfernt Flipflops und bewahrt gleichzeitig die ursprüngliche Funktionalität, indem die zeitlichen Eigenschaften der Schaltung entsprechend angepasst werden. Die entfernten Zustandsflipflops können nicht mehr durch sequentielles Reverse Engineering von Netzlisten auf Gatterebene identifiziert werden, was ein erfolgreiches sequentielles Reverse Engineering verhindert. Da Timing Camouflage nur auf Flipflops ohne kombinatorischen Rückkopplungspfad angewendet werden kann, werden in der Arbeit zusätzlich zwei Techniken entwickelt, die einen Zustandsautomaten so umgestalten, dass ein kombinatorischer Rückkopplungspfad für das Zustandsflipflop, das durch Timing Camouflage entfernt werden soll, vermieden wird. Die zweite Obfuskationstechnik nutzt die hergeleiteten, häufig auftretenden und ausnutzbaren Merkmale von Methoden zur Identifizierung von Zustandsflipflops. Sie fügt eine—für das Identifizierungsverfahren von Zustandsflipflops hochattraktive—Honeypot-Zustandsmaschine hinzu und übersetzt die originale Zustandsmaschine in eine—für das Identifizierungsverfahren von Zustandsflipflops höchst unattraktive—Zustandsmaschine, ohne ihre ursprüngliche Funktionalität zu verändern. Eine Technik zur Identifizierung von Zustandsflipflops wird nun den attraktiven Honeypot-Zustandsautomaten und nicht den unattraktiven, originalen Zustandsautomaten als besten Kandidaten für einen Zustandsautomaten identifizieren oder den korrekten, originalen Zustandsautomaten überhaupt nicht mehr identifizieren können. Dies erschwert oder verhindert ein erfolgreiches sequentielles Reverse Engineering.

Abschließend beschäftigt sich die Arbeit noch mit möglichen negativen Folgen, die von Schutzmechanismen verursacht werden können. Die Arbeit beschreibt einen Fehlerangriff, bei dem die Schutztechnik Logic Locking verwendet wird, um Fehler in eine Netzliste auf Gatterebene einzufügen. Logic Locking fügt schlüsselgesteuerte Locking Gatter in eine Netzliste ein. Dies stellt sicher, dass die Schaltung nur dann korrekt funktioniert, wenn der richtige, geheime Locking Schlüssel angelegt ist. Allerdings entspricht das Ändern eines Locking Schlüsselbits einer Fehlerinjektion. Die Arbeit analysiert die Verwendbarkeit dieses eingefügten Fehlers basierend auf der eingesetzten Locking Technik, Schlüsselverwaltung und Fehleranalysetechnik. Schließlich demonstriert ein Anwendungsfall den durch Logik Locking ermöglichten Fehlerangriff, indem ein geheimer, kryptografischer Schlüssel extrahiert wird.

Acknowledgment

I would like to thank my supervisor, Prof. Dr.-Ing. Georg Sigl, for the opportunity to do a Ph.D. at the Technical University of Munich and for his constant support.

I would like to thank my colleagues who have always been at my side with advice and support and who have become real friends over time.

I would like to thank all co-authors for the great cooperation and productive discussions.

I would like to thank all the students I had the pleasure to supervise during my Ph.D. for their input and contribution to my research topic.

I would like to thank my family and friends for their support. I would especially like to thank my husband Felix, who accompanied and encouraged me through all phases of my Ph.D.

Contents

1 Introduction

Chips rule our world.

Nowadays, nearly every part of our life relies on the functionality of chips: smart homes, cars, mobile phones, computers, medical equipment, safety locks, military, airplanes, and more. To provide this mass of required chips, many chip production foundries were built all over the world. In 2021, China and Taiwan were the leading countries in new semiconductor fabrication projects worldwide, see Figure 1.1.

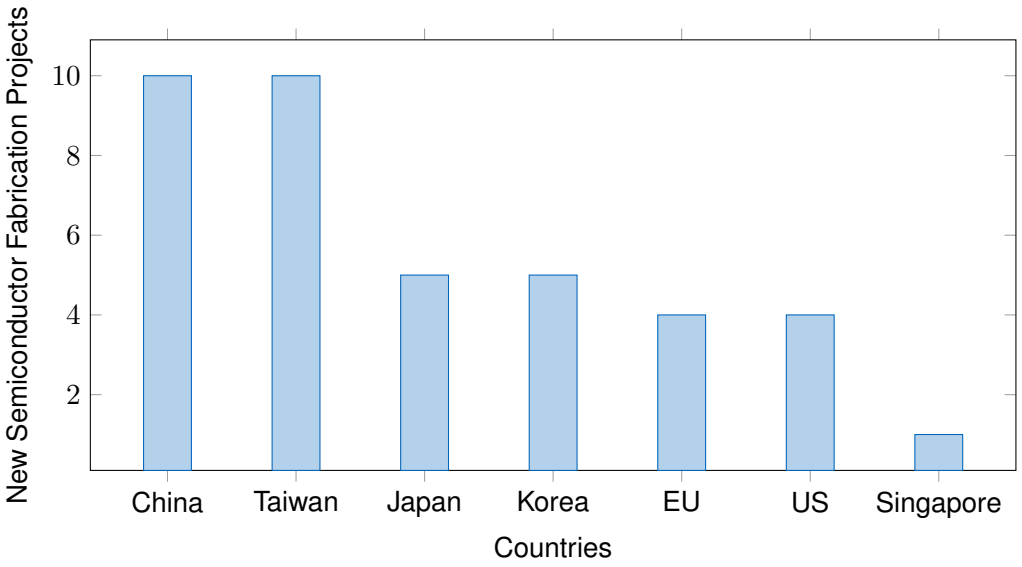


Figure 1.1 Statistic of new semiconductor fabrication projects 2021 [SIA22]

Though, the focus on third-party companies and countries for taking tasks in chip production holds a risk for chip designers. It becomes very challenging for them to avoid malicious modification or the illegal overproduction of their products or to verify the unmodified production of their end products. This dissertation contributes to these searing challenges by taking new perspectives on sequential Reverse Engineering (RE) and obfuscation.

The term chip is used as a synonym for the term Integrated Circuit (IC). Both terms indicate a single module that consists of an electronic circuit on a semiconductor material [Utm21]. The electronic circuit is composed of transistors and other elements, like resistors or capacitors, which are all connected with electronic wires. For extended functionality, new, more complex chips can be produced, or different chips and components can be combined on a so-called Printed Circuit Board (PCB). To produce a chip, a large number of standardized design and production steps, like the Register Transfer Level (RTL) code generation, logic and physical synthesis, or fabrication and testing, must be performed, see Section 2.1.

The term chip itself can be further classified [Utm21]. Some chips only fulfill a specific purpose: a customized one, called Application Specific Integrated Circuit (ASIC), or a standardized one, called Application Specific Standard Product (ASSP). Both types do not necessarily

include a processor. Instead, a System on Chip (SoC) describes a complete system, including a processor or, for example, also memory or peripherals. Examples for SoCs are microprocessors or microcontrollers. The last type of chips are programmable circuits, called Field-Programmable-Gate-Arrays (FPGAs). The standard FPGA does not contain a processor, while the extended version, called SoC FPGA, again describes a complete system, including a processor, memory, peripherals, and an FPGA.

This dissertation concentrates on ICs and in particular on ASICs, ASSPs, and SoCs.

1.1 Motivation

RE can be a risk and a chance for the present challenges of foreign chip production [Qua+16; Kum00]. It aims to extract the chip's design. Depending on the provided entry point and RE technique, this includes, for example, the extraction of the memory content, the gate-level netlist, or the chip's application purpose [Qua+16; TJ09], see Chapter 2. Overall, the result of RE is always an increased understanding of the chip in hand.

Risk: If RE is used by an attacker, he/she can gain further knowledge about the chip. This can be a risk to the security of the chip [e.g. Qua+16; TJ09; Kum00]. In the following, four exemplary scenarios of a security risk are listed. However, this list could be further continued.

- If the attacker reveals the presence of a specific crypto algorithm, he/she could apply suitable attacks to extract the secret key [e.g. Qua+16].
- If the attacker reverse engineers the design of his/her competitive product, he/she could gain a benefit over his/her competing company [e.g. Kum00].
- If the attacker identifies signals and wires that carry secret information on the chip, he/she could use them to insert a leakage Hardware Trojan (HT) or to apply probing attacks [e.g. TJ09].
- If the attacker reverse engineers the content of a read-protected memory, he/she could extract secret data that is stored on the chip [e.g. Qua+16].

Chance: If RE is used by a chip owner, he/she can verify its own products or countermeasures [e.g. Qua+16; LBL21; Lip+20]. After receiving the final products from the third-party manufacturing, testing, or packaging factory, the chip owner can reverse engineer a small number of the received chips and verify if their functionality or design is equal to the intended functionality or design. In addition, chip owners or chip designers who want to develop effective countermeasures against the above-described RE risks must test their protection mechanisms by exposing them to RE techniques. Further, a chip owner can use RE to uncover patent infringement and cloned products [e.g. Kum00].

Depending on the RE technique, its target includes specific intermediate results of the chip's functionality and design [Azr+21; AGM19]. A target that is often of special interest to an attacker or to a chip owner is the control logic of a design or, in particular, the Finite State Machine (FSM) of a design. The importance of an FSM comes due to three main reasons.

1. FSMs are often the Intellectual Property (IP) of a chip which consumed the highest costs and time to develop improvements that prevail against competitors. While many chip components are standardized, like adders or multipliers, the FSM is usually individual for the design [AGM19]. Thus, FSMs are interesting targets for competing companies or IP theft purposes.

2. FSMs often have comprehensive power over the chip's functionality. Thus, understanding their structure can enable attacks tailored to the implemented algorithm. To show the range of possible attack scenarios, three examples are given:
    - If the attacker reverse engineers the correct FSM of a cryptographic algorithm, he/she can conclude what cipher is used and how its logic is implemented. This knowledge supports the exploit of known vulnerabilities of a cipher, like extracting the secret key of an Advanced Encryption Standard (AES) [DR02] by skipping all but one AES round with injected faults [CT05].
    - If the attacker reverse engineers the correct FSM of an access control mechanism, he/she can identify the correct position and point in time to inject a fault to bypass the access control check.
    - Understanding the FSMs' structures can enable effective insertion of HTs. HTs which are inserted by using the design's FSM can make use of the FSM's comprehensive power over the functionality of the chip. For example, the attacker can design an HT which skips all but one round of the AES cipher when triggered. This eases the AES key extraction significantly [CT05].

      In addition, the attacker can exploit unused states of an FSM to hide the HTs in a design [DQ14]. The work in [WP13] developed an automatic HT insertion tool which triggers HTs with unused FSM states.

3. Sophisticated HTs are often not only attached to the FSM of a design, but also implemented by an FSM itself [Wan+11]. A chip owner can reverse engineer his/her manufactured product's FSM(s) to verify that no HT was inserted and that the original FSM(s) were not manipulated [MZJ16].

These reasons show that RE and obfuscation techniques that concentrate on FSMs of a design, i.e. sequential RE and sequential obfuscation, deserve special interest. This dissertation contributes to the topic of *hardware gate-level netlist sequential RE and FSM obfuscation*.

Gate-level netlist RE analyzes a gate-level netlist to extract knowledge about the implemented design [e.g. Sub+13], see Chapter 2. A gate-level netlist describes a design using combinational gates, like AND, OR, and Inverter gates, and sequential components, like synchronous or asynchronous Flip-Flops (FFs), from a technology library [Utm21]. The netlist elements are connected with wires and driven by inputs to produce specific outputs.

Sequential gate-level netlist RE concentrates on the combinational and sequential gates in the netlist that belong to the FSM, including the FFs which hold the FSM state, so-called State Flip-Flops (sFFs) [e.g. McE01], see Chapter 3. Netlist elements that belong to an FSM perform a design task quite different from other design tasks. FSM netlist elements determine the next state bits of an FSM [Ped13] what is usually no standardized procedure but individual for each FSM and FSM bit. On top of that, the next state bit determination can depend on various design variables, like previous state bits, control or data design variables, or primary design inputs. All this also affects the properties of sFFs in a gate-level netlist. Sequential gate-level RE makes use of all known differences between FSM netlist elements and other netlist elements to identify and interpret FSM netlist elements [e.g. Mea+16; Fyr+18]. Thus, it considers the differences in functional and structural terms.

FSM obfuscation includes obfuscation techniques which aim to protect FSMs or sFFs [e.g. HP20], see Chapter 5. We exclude techniques that have an effect on the security of FSMs or sFFs but do not have this as a primary goal; instead, the effect occurs as a side effect, for example.

1.2 Objectives

The objectives to which this thesis contributes can be summarized as follows:

- **Definition of FSM components in RTL and netlist:** There exist mathematical descriptions of how to define an FSM [Moo56; Mea55], but there is a lack of clear definitions of what belongs to an FSM in an RTL description or in a gate-level netlist of a design. The works in [Fyr+18] or [RB22] for example state what FFs they assume as FFs belonging to an FSM, while the work in [Gei+20] introduces three FF quality levels based on how much they affect the design's functionality. However, their approaches also lack explicit and conservative formulations. Clear definitions of what belongs to an FSM are figured out to be not straightforward, as control logic can be implemented in several different ways. The RTL description solely gives one possible implementation selected by the designer, while the gate-level netlist is based on this designer's decision and on several further synthesis optimization decisions. However, uniform definitions are highly important to evaluate sequential RE methods and enable a fair comparison of different evaluation results [Azr+21].

- **Analysis of FSM gate-level netlist features:** Different sFF identification techniques use different features to identify sFFs in a gate-level netlist [e.g. McE01; Mea+16]. Most authors reason about their choice of features for their proposed techniques. However, this argumentation mostly focuses on the majority of FSM structures. As already mentioned, there is a large variety of how to implement a certain control logic. If the features are adapted to the majority of FSM implementations, implementations with rare characteristics will be ignored. There is a lack of work about understanding FSM gate-level netlist properties in a more generalized and detailed approach.

- **Improving sFF identification:** To extract FSMs one first have to identify sFFs in a gate-level netlist [McE01]. However, to the best of our knowledge, all existing sFF identification methods are heuristic approaches, and so—in contrast to other FSM extraction steps—no perfect solution exists yet for identifying the correct sFFs [Azr+21]. Thus, it is a continuous objective in literature to improve sFF identification techniques to achieve higher accuracy.

- **Protecting FSMs:** Similar to other hardware protection mechanisms, also the protection of FSMs against RE attacks faces a back and forth of new obfuscation methods [KJC19; LNO21] and new attacks [Sah+21; Rah+23]. Thus, also the task of developing the not breakable protection mechanism is a continuous objective in literature. Most existing protection mechanisms use a secret sequence of bits to prevent the RE of the correct FSM functionality [e.g. LNO21]. So, there is a need for novel, innovative ideas for protection techniques that are not primarily based on the secrecy of bits. This includes, but is not limited to, combining different mechanisms beneficially or exploiting not yet used circuit characteristics.

- **Analysis of risks of using protection mechanisms:** A rather unnoticed objective in literature is understanding what consequences a (combined) protection technique can have if it has not been fully thought through. Protection mechanisms introduce new elements into an RTL design or into a gate-level netlist [e.g. RKM08a], what might open new attack possibilities. The more different protection techniques are applied, the higher the risk of an accidentally introduced new attack vector, because it is more likely that one of the used techniques makes an attack possible.

1.3 Contributions

The contributions of the thesis are predominantly based on the following author's publications:

- Michaela Brunner et al. "Logic Locking Induced Fault Attacks". In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 114–119. DOI: 10.1109/ISVLSI49217.2020.00030 [Bru+20]
- Michaela Brunner et al. "Toward a Human-Readable State Machine Extraction". In: *ACM Trans. Des. Autom. Electron. Syst.* 27.6 (June 2022). ISSN: 1084-4309. DOI: 10.1145/3513086. https://doi.org/10.1145/3513086 [Bru+22a]
- Michaela Brunner et al. "Timing Camouflage Enabled State Machine Obfuscation". In: *2022 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 2022, pp. 1–7. DOI: 10.1109/PAINE56030.2022.10014810 [Bru+22b]
- Michaela Brunner et al. "Hardware Honeypot: Setting Sequential Reverse Engineering on a Wrong Track". In: *arXiv preprint arXiv:2305.03707* (2023). Peer-reviewed version: *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*. 2024, pp. 47–52. DOI:10.1109/DDECS60919.2024.10508924. [Bru+23]

The thesis contributes to the above-described objectives as follows:

- **Contribution 1 (Classification of State Flip-Flop Sets (sFF sets)):** The thesis investigates the degrees of freedom of the mathematical description for FSMs. Based on this analysis, new sFF and sFF set properties are derived, which are used to classify Flip-Flop Sets (FF sets) into sFF sets and Non State Flip-Flop Sets (nFF sets). The properties are further used to allow an even more fine-grained classification of sFF sets into newly derived sFF set property categories. In addition, also sFF identification algorithms can be classified using the proposed categories.
- **Contribution 2 (sFF set definition for a human-readable state machine):** The thesis presents a novel definition for an sFF set, the so-called sFF set definition for a human-readable state machine. A human-readable state machine reflects the actual state machine and is not overloaded with excessive extra design information. For this purpose, the definition also considers the newly derived sFF and sFF set properties. According to these findings, the definition includes all sFF sets which are understood as a reasonable FSM and excludes all sFF sets which are understood as no or multiple FSMs.
- **Contribution 3 (Feature analysis of sFF identification methods):** The thesis identifies common and exploitable features of existing sFF identification methods. Different sFF identification methods use different features to identify sFFs. However, there are some features that are more frequently used than others or features that are—although not identical—comparable to each other.
- **Contribution 4 (Post-processing of sFF sets):** This thesis develops four post-processing algorithms to improve the output of sFF identification methods. The post-processing is based on the previously developed classification categories. The post-processed sFF set fulfills the targeted sFF properties and is, thus, considered to be a reasonable FSM with respect to the sFF set definition for human-readable state machines. The results show that post-processing largely improves the outcome of sFF identification algorithms which are mainly based on similarity-based features.

- **Contribution 5 (Novel FSM obfuscation based on camouflaging):** The thesis develops a novel FSM obfuscation technique which is not based on a secret locking key—as this is the case for most FSM obfuscation methods—but on a camouflaging technique, called Timing Camouflage [Zha+18b]. However, sFFs often have combinational Feedback Paths (FPs), which makes it challenging to use Timing Camouflage directly. Thus, we developed two methods that redesign an FSM with the following objective: at least one of the FSM's FFs is free of a combinational FP and the actual functionality stays the same. The results demonstrate that this novel obfuscation methodology prevents current techniques from extracting the correct state machine.
- **Contribution 6 (Novel FSM obfuscation based on misleading tracks):** The thesis develops a second novel FSM obfuscation. While the first one is based on a camouflaging technique, the second idea exploits the assumptions of RE algorithms. The obfuscation comprises two parts: 1) Honeypots lead the RE tools to a false but for the tools very attractive state machine, and 2) the attractiveness of the actual state machine is reduced. The results show that state-of-the-art RE methods choose the honeypot as the best state machine candidate or no longer discover the correct, original state machine.
- **Contribution 7 (Enabled fault attacks with logic locking):** The thesis analyzes logic locking algorithms from the point of view of introducing new attack possibilities. Logic locking introduces secret locking key bits into a gate-level netlist. Typically, these bits are secured against read-out attacks but not necessarily against modification attacks. However, modifying one locking key bit enables fault attacks. The thesis analyzes the applicability of this risk based on different logic locking, key management, and fault analysis techniques. In addition, the thesis demonstrates the threat by extracting a secret AES key byte.

The author further contributes to the following publications:

- Michaela Brunner et al. "Improving on State Register Identification in Sequential Hardware Reverse Engineering". In: *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2019, pp. 151–160. DOI: 10.1109/HST.2019.8740844 [BBS19] to improve the performance of an existing sFF identification method. This paper is based on parts of the author's research internship [Bru17] and master thesis [Bru18], but its content was revised and extended during the author's Ph.D. studies.
- Grace Li Zhang et al. "TimingCamouflage+: Netlist Security Enhancement With Unconventional Timing". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4482–4495. DOI: 10.1109/TCAD.2020.2974338 [Zha+20b] and Grace Li Zhang et al. "Timing Resilience for Efficient and Secure Circuits". In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 623–628. DOI: 10.1109/ASP-DAC47756.2020.9045352 [Zha+20a] to support the security point of view for this novel camouflaging technique.
- Matthias Ludwig et al. "CRESS: Framework for Vulnerability Assessment of Attack Scenarios in Hardware Reverse Engineering". In: *2021 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 2021, pp. 1–8. DOI: 10.1109/PAINE54418.2021.9707695 [Lud+21] to develop a novel framework to evaluate RE enabled attacks.
- Johanna Baehr et al. "Open Source Hardware Design and Hardware Reverse Engineering: A Security Analysis". In: *2022 25th Euromicro Conference on Digital System Design*

(DSD). 2022, pp. 504–512. DOI: 10.1109/DSD57027.2022.00073 [Bae+22] to add the sequential RE point of view for open-source effects in RE.

- Bernhard Lippmann et al. “VE-FIDES: Designing Trustworthy Supply Chains Using Innovative Fingerprinting Implementations”. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137026 [Lip+23] to put the Timing Camouflage enabled FSM obfuscation in the context of protecting fingerprinting systems.

However, these publications are outside the scope of the contributions of this thesis.

1.4 Outline

The thesis is organized as follows: Chapter 2 gives an introduction to the hardware RE procedure, including required steps and possible analysis results. Next, the above-introduced contributions are presented: Contribution 1 is shown in Sections 3.1 to 3.4.1, in Section 3.5, and in Section 4.2. Contribution 2 is presented in Section 3.4.2 and in Section 4.2. Contribution 3 is given in Section 3.5. Contribution 4 is presented in Chapter 4. Contribution 5 is shown in Section 5.2. Contribution 6 is given in Section 5.3. Contribution 7 is presented in Chapter 6. The thesis concludes by summarizing the shown strengths of exploring hardware gate-level netlist features and of combining different RE or different obfuscation techniques.

2 Hardware Netlist Reverse Engineering

This chapter gives an overview of hardware gate-level netlist RE. Thus, first, the chip life cycle is presented because hardware gate-level netlist RE reverses parts of the chip life cycle. Then, the two main parts of hardware gate-level netlist RE are explained: physical gate-level netlist RE and functional gate-level netlist RE.

The netlist path definitions have already been pre-published in the author's works [Bru+22a].

RE in general is a huge field, including software RE [e.g. CDC11], bitstream RE [e.g. Zha+19], or hardware RE [e.g. Qua+16]. This thesis concentrates on hardware RE, which again includes many different aspects, like the RE of memory content and structures, the optical inspection of PCB components, probing, simulation, or the extraction and static analysis of chips' gate-level netlists [Qua+16; TJ09; Sto+21]. Every aspect is a separate research field and has different RE targets. However, they might also be combined to improve the overall results of increasing the understanding of the product's functionality. This thesis concentrates on the static analysis of chips' gate-level netlists.

2.1 Chip Life Cycle

The gate-level netlist of a chip is a side product of the chip production process. To produce a chip, many design and production steps have to be fulfilled first [Utm21], see Figure 2.1. The chip's life cycle starts with an idea for its functionality, which is noted down with a specification in the design house. Second, the architecture options are determined, and third-party IP is ordered. The resulting system, including specification and architecture options, is verified against the desired functionality. Next, the specification is implemented with an Hardware Description Language (HDL) like Verilog or VHDL, resulting in an RTL source code. Again, the result is verified, e.g. by functional simulation. The RTL code is now synthesized by a logic synthesis, which translates the RTL code description into a gate-level netlist description using a provided technology library. The gate-level netlist contains combinational gates, like AND, OR, Inverter gates, sequential components, like synchronous or asynchronous FFs, inputs, outputs, and wires. The applied technology library defines what gates are available during the synthesis. Thus, depending on the library only basic gate types, like AND, OR, Inverter, or also more sophisticated gate types, like adders or multiplexers, could be available. The technology library also defines characteristics for each gate, e.g. regarding timing or power. This and other timing-relevant information is used during the subsequent static timing analysis. Next, additional logic is added, which is not required for the desired functionality but for later testing possibilities. Before performing the next synthesis step, the netlist is verified once more. The second synthesis step, the physical synthesis, places and routes the gate-level netlist, what results in a layout netlist. After a final verification, all required data for production is handed over to the foundry in the form of GDSII data. The foundry now produces the chip based on the provided GDSII data. Next, the chip is tested, either by a dedicated testing factory or by the design house itself, and then packaged. Finally, the end products are sold to the end users.

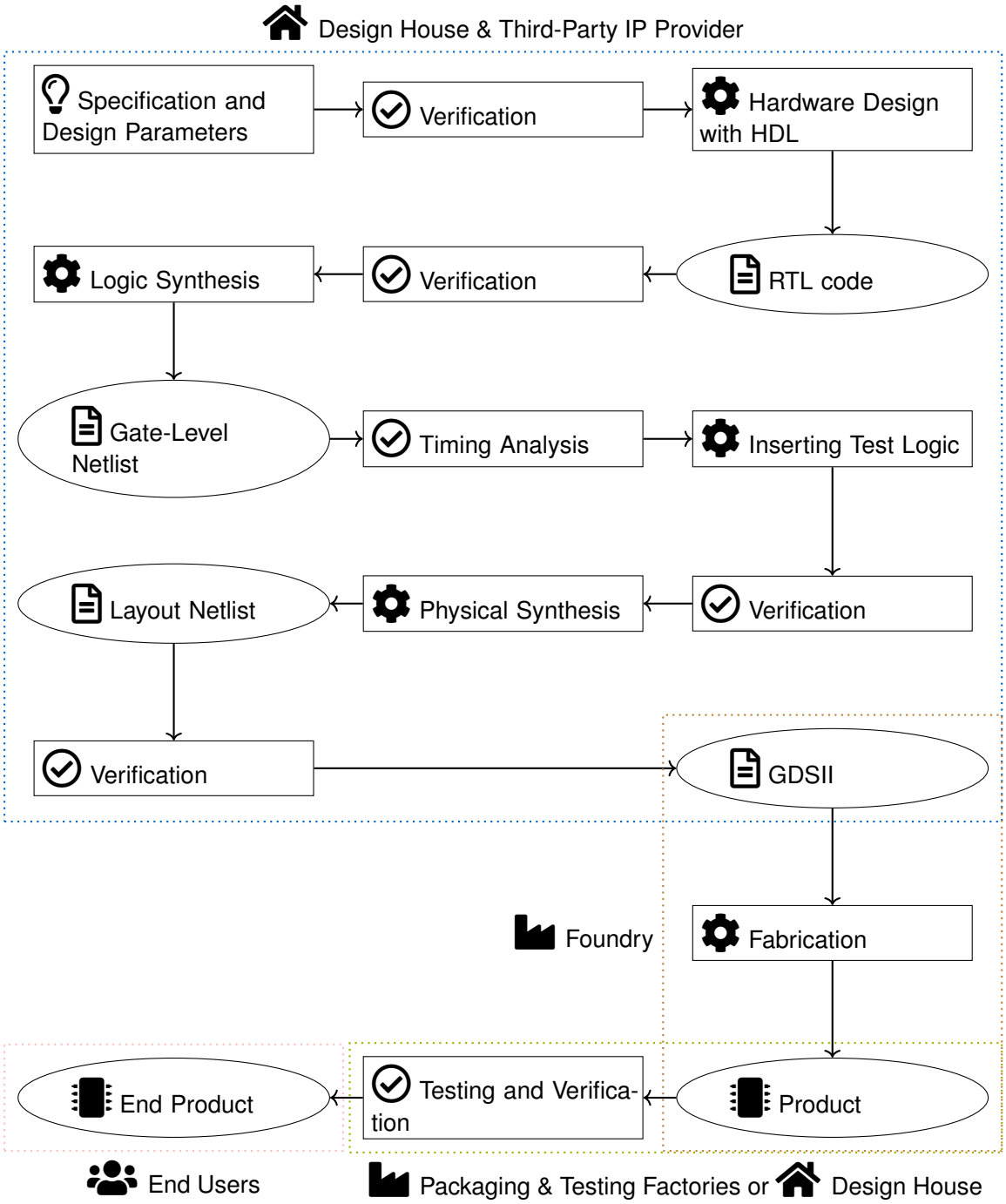


Figure 2.1 Chip life cycle showing the single steps from the initial chip specification to the final end product, and the engaged parties [Utm21]. Icons provided with [Kru].

The chip life cycle in Figure 2.1 shows that depending on who the attacker is, different steps have to be performed to receive the gate-level netlist of a design, which is the starting point for functional gate-level netlist RE [Azr+21]. The end user and packaging or testing factories only have access to the manufactured product. Thus, they first have to perform some extra steps, so-called physical gate-level netlist RE (Section 2.2), to obtain the gate-level netlist. Instead, the manufacturing foundry has access to the GDSII data, which can easily be translated to a gate-level netlist, as it contains all required information. The design house or IP provider directly has access to their own design RTL codes, gate-level and layout netlists.

2.2 Physical Gate-Level Netlist Reverse Engineering

Physical gate-level netlist RE denotes the process of RE a manufactured product, i.e. a chip, to its gate-level netlist. Typically, the product equals the end product, i.e. the last step of Figure 2.1, which means that the *attacker is the end user*. To reach the gate-level netlist, the attacker has to reverse the fabrication steps that are done by the packaging and manufacturing factories.

The steps for physical gate-level netlist RE are shown in Figure 2.2 and described in the following [TJ09; LBL21; Qua+16; Lip+20]:

1. **Depackaging:** To perform all further steps, first, the reverse engineer has to remove the package of the chip. Depending on the package, this can be achieved by chemical etching or mechanical or thermal processes. Important is that this step does not damage the chip surface under the package.

2. **Delayering:** State-of-the-art chip production uses technology sizes down to a few nanometers and several metal layers that contain wires and vias. A wire connects two positions on the same metal layer, while a via connects two positions on two neighbor metal layers. Thus, for the depackaged chip, all available metal layers have to be removed one by one until the polysilicon transistor gate layer is reached. The delayering is a challenging process that varies depending on the material used for the chip layers, the layer thicknesses that differ for each layer, or the technology that is planned for the next physical RE step. Typically, the delayering is achieved by a combination of different techniques, like plasma or wet etching, or polishing. In general, the more the technology size shrinks and the more metal layers are used, the more challenging the delayering process becomes. For each layer, the reverse engineer has to ensure that the exact thickness of a layer is removed. If too little or too much material was removed, one does not see the wires of the expected metal layer but of the higher or lower metal layer. In addition, the reverse engineer has to ensure that the delayering is done precisely planer. Otherwise, one sees a mixture of wire fragments of different metal layers. If the precision of the delayering is not sufficient for one layer, the process has to be repeated with a new chip.

3. **Imaging:** Each layer of the delayered chip is imaged with an Scanning Electron Microscope (SEM) because the resolution of optical imaging is not high enough for small technology sizes. The image of an SEM is much smaller than the complete chip size. Thus, the SEM makes multiple images of the same chip layer. For this, the SEM is positioned automatically, such that it reaches each part of the chip layer once.

4. **Stitching and Alignment:** The image pieces of one chip layer have to be stitched to retrieve an overall picture of the layer. All 2D-stitched layers must then be aligned,

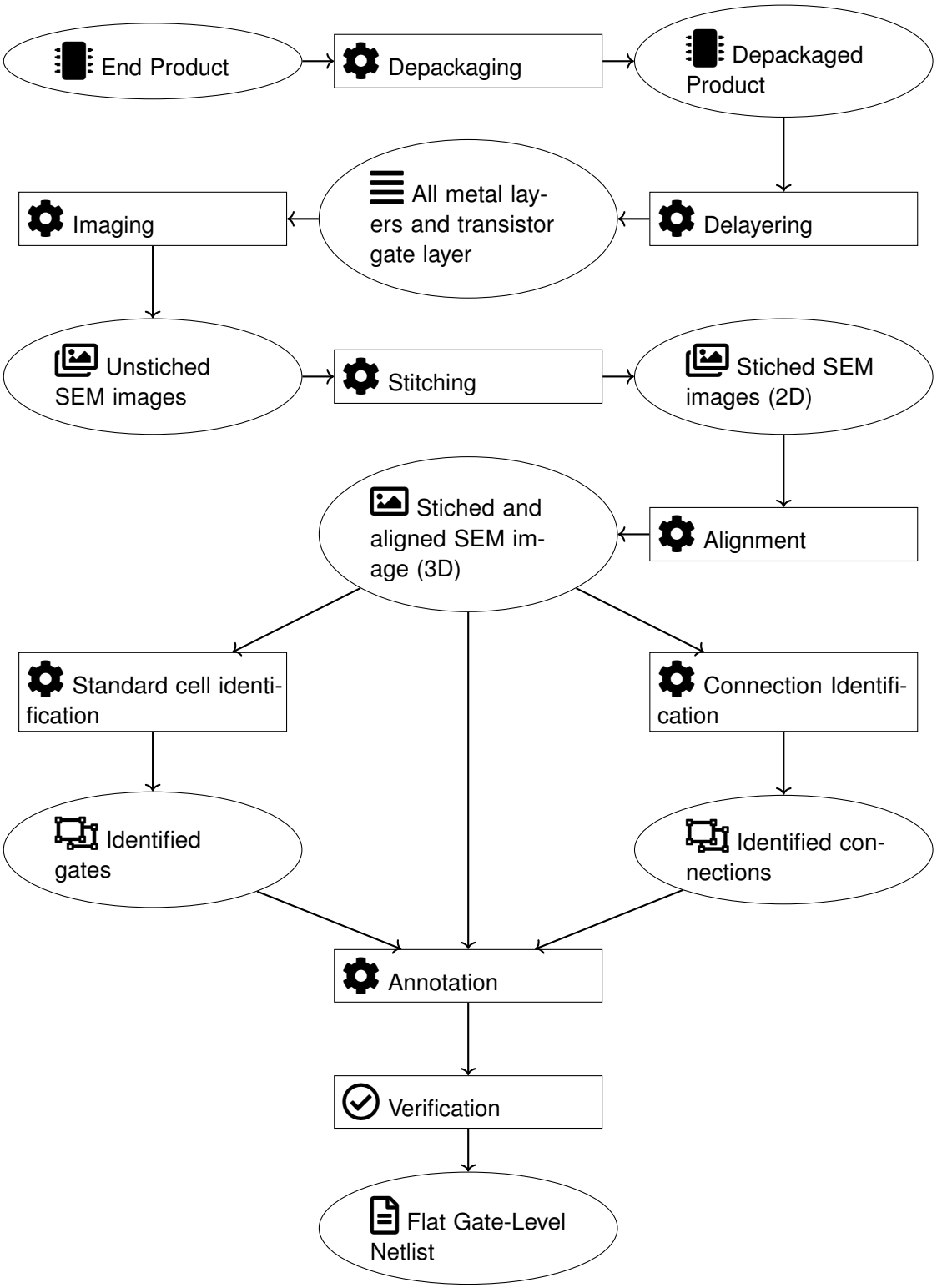


Figure 2.2 Physical gate-level netlist RE workflow [TJ09; LBL21; Qua+16; Lip+20]. Icons provided with [Kru].

resulting in a 3D SEM image. Depending on the image quality, the stitching and alignment process can be more or less challenging. There exist algorithms that perform 2D stitching and 3D alignment [e.g. SLG21].

5. **Annotation:** The 3D SEM image is converted to a gate-level netlist. On the polysilicon transistor gate layer, gates are located and matched to a library. Image processing and machine learning can support the detection and labeling of gate structures, wires, and vias. Identified gates, wires, and vias are then translated to a flattened, i.e. not hierarchical, gate-level netlist.

6. **Verification:** Finally, some simple verification is used to check the plausibility of the labeled SEM image and of the resulting flattened gate-level netlist, e.g. if there are shortcuts, dangling wires, or unconnected inputs or outputs. As each step of the physical gate-level netlist RE is highly error-prone, this last verification step is necessary to remove obvious errors. However, still, the resulting netlist might not be completely error-free.

If the *attacker is the packaging or testing factory*, all steps except step 1. for physical gate-level netlist RE are required to reach the gate-level netlist. In addition, a packaging or testing factory might have access to more information about the chip, like testing vectors, which could ease some of the remaining physical gate-level netlist RE processes.

Physical gate-level netlist RE is no longer required if the *attacker is the manufacturing foundry or the design house or IP provider*. Such an attacker already has access to the gate-level netlist directly (design house or IP provider) as they do the synthesis by themselves or can easily translate the available data to a gate-level netlist (manufacturing foundry) as they have access to the GDSII data which contain the gate-level netlist together with layout data.

2.3 Functional Gate-Level Netlist Reverse Engineering

Functional gate-level netlist RE denotes the process of RE the gate-level netlist to the design's functionality, i.e. the analysis of the gate-level netlist [Azr+21; AGM19; Qua+16]. Thus, the target of functional gate-level netlist RE is to increase the understanding of the design and abstract the netlist information to a higher level. While physical gate-level netlist RE predominantly consists of consecutive steps, the steps or algorithms of functional gate-level netlist RE are often not clearly ordered but can also be performed in parallel [Lud+21]. Depending on the type or amount of information an attacker wants to derive from the gate-level netlist, one or multiple steps or algorithms are applied [Azr+21].

In the beginning, when the complexity of chips and of their design process was still small, RE was mainly dependent on the skills of RE experts [TJ09; HYH99]. They used available information, like technical manuals, patents, or simulation. A decisive factor was the experience a reverse engineer had. An experienced reverse engineer might e.g. spot logic structures in a gate-level netlist he/she already reverse engineered once before. Over time, the complexity of chips and of designing chips increased. Thus, RE tools were developed or improved. Nowadays, functional gate-level netlist RE relies more and more on automated algorithms [Azr+21]. Some researchers also make their developed frameworks openly available [e.g. MZJ19; Emb19]. Despite this, the experience of reverse engineers still plays a role [Fyr+17; Wie+19]. They, for instance, select the algorithms to run, decide about good parameters for running the algorithm, or evaluate the RE result. Functional RE algorithms make use of existing techniques and methods in graph theory because a gate-level netlist can easily be translated to a graph representation [Azr+21]. The gate-level netlist consists of standard cells. Standard

cells include clock-controlled cells, also known as FFs, and gates out of the set of combinational gates $\mathbb{N}$. What type of FFs and combinational gates exist, is defined in the standard cell library, which is applied during the logic synthesis [Utm21]. Each input and output of a standard cell is assigned to a wire, an input or output. Such, the standard cells connect with each other and form a netlist. When translating the gate-level netlist to a graph, standard cells translate to graph nodes, and wires translate to edges between nodes [Azr+21]. Inputs and outputs stay unchanged. Typically, a netlist is translated to a directed graph. For a directed graph, edges represent not only a wire between two standard cells but also the direction of the wire, i.e. if the wire connects to an input or output of a cell. Directed graphs allow the definition of paths. Correspondingly, a path can also be defined for netlists.

Definition 1 (Netlist Path) *A path $p(a, b)$ in the netlist is a connected sequence of standard cells $(a, g_1, \ldots, g_l, b)$, such that logic information flows from the standard cell a to the standard cell b. Each standard cell occurs only once in a path. This is equivalent to the definition of simple paths in graph theory.*

To denote the available connections, i.e. paths, in the netlist, the function $P(a, b)$ is introduced.

Definition 2 (Netlist Path Function) *The output of function $P(a, b)$ is true if a path $p(a, b)$ exists from a to b, and false otherwise.*

In the following, different intermediate results which are achieved by the algorithms of functional gate-level netlist RE are shortly introduced [Azr+21; AGM19; Lud+21]:

- **Netlist partitioning:** Although there is no fixed order in how algorithms are applied, netlist partitioning is usually one of the first steps in functional gate-level netlist RE. As the name indicates, netlist partitioning divides a large netlist into smaller subparts called partitions. However, this does not necessarily include the translation of a flattened netlist into a hierarchical netlist [Web+22]. The resulting smaller partitions of the netlist can be analyzed separately in a second step, what can reduce the complexity of algorithms significantly. Thus, partitioning algorithms should split the netlist into logical related components or modules so that the consecutive algorithms achieve good results when they are applied to the partitions.

  There exist various approaches to partitioning a netlist.

  One approach uses the connectivity between partitions. Some algorithms of this approach partition large netlists by identifying partitions that are connected with a small number of edges using a method based on the *normalized cut* [e.g. Cou+16]. Others partition large netlists by extending very small partitions with closely connected partitions. They use the clustering method *shared nearest neighbor* [e.g. Azr+16] or neighboring partitions which increase the so-called *modularity*, a measure for the connectivity within partitions [Wer+18]. Methods of this approach are based on known graph partitioning algorithms, which, for example, allows a reverse engineer to use already implemented algorithms in programming language packages, like *scikit-learn* [Ped+11].

  Another approach uses functional affiliation to partition a netlist. Most methods of this approach aim at the identification of words in a netlist and then partition the netlist based on the identified words [e.g. Mea+18]. More information on words and word identification is given under the bullet point 'data path identification'.

  As in physical RE, also for functional gate-level netlist RE, machine learning methods nowadays support or improve RE. In [Hon+22] and [Alr+22], Graph Neural Networks (GNNs) are used to improve netlist partitioning. Currently, GNNs are the preferred choice

for supporting RE algorithms. They classify nodes or edges or determine the functionality of nodes or edges depending on their neighbor nodes or edges. This naturally supports the mode of operation of former netlist partitioning approaches. The method in [Alr+22] applies the GNN on a feature vector, which is extracted from the netlist graph and includes gate types or the number of inputs and outputs. Instead, the method in [Hon+22] uses the GNN to minimize a loss function, which is based on the *normalized cut*.

- **Data path identification:** An identified data path can significantly increase the understanding of the analyzed gate-level netlist. The identified data path shows how the data propagates through the circuit, how data words interact with each other, and what sizes the data words of a design have. A data word is defined as a group of signals that propagates a related set of data bits through a netlist. Data bits are defined to be related if they, for example, belong to a 16-bit data register which is added to another 16-bit data register. Whenever a word is identified, this word can be further used to find the succeeding or proceeding word in a netlist, resulting in a data path in the end [Sub+13].

  Again, there exist different approaches to identifying words. Some methods concentrate on one of these two approaches, while others use both.

  One approach analyzes the netlist structures of subgraphs that determine certain signals in a netlist. The method in [Alb+20] analyzes the predecessors and successors of FFs in an FF dependency graph, e.g. which common and which different preceding and succeeding FFs or control signals a FF has, to determine data words and the corresponding data flow. Other algorithms produce their results using the netlist structure of only one subgraph, typically the preceding subgraph. The more similar two netlist structures of two signals' subgraphs are, the more likely these two signals belong to the same word. To determine the similarity of signals, methods calculate a customized similarity score [Mea+18] or a hash value [Li+13] based on gate types.

  Another approach analyzes the netlist's functionality of subgraphs that determines a data word in a netlist. The method in [YC16] determines Boolean functions out of the gate-level netlist and searches them for binary words, for example, a shifter, adder, or multiplier. Also, the method in [Sub+13; Sub+14] uses Boolean functions to identify words. However, the work concentrates on partitions with a specific number of inter-partition connections. These partitions are then classified into equal Boolean functionalities, and suitable partitions of the same class are aggregated. Finally, all signals which reach and leave the partition are defined as words. The method in [Li+13] uses function recovery to verify word candidates that were identified with a structural approach first.

- **FSM identification:** In parallel to the data path identification, one can aim for the identification and extraction of FSMs, also called *sequential netlist RE*. In contrast, all other functional netlist RE steps are summarized under the term *combinational netlist RE*. As discussed in the introduction in Chapter 1, the knowledge about the design's functionality is an interesting target for both, the attackers and the chip owners. Thus, the thesis contributes to this topic.

  There exist two main approaches to identifying an FSM in a gate-level netlist.

  One approach splits the identification of netlist gates that belong to the FSM into two steps: 1) the identification of FFs which belong to the FSM and 2) the identification of combinational gates which belong to the FSM [e.g. McE01]. While for the FSM's FF identification step, a large number of various heuristic approaches exist, the second step, the identification of combinational FSM gates, is based on the results of the first step and can be considered solved if the first step achieves the correct sFF set. An open

research question regarding the identification of combinational FSM gates is, for example, its potential to compensate for errors of the previous step. A detailed analysis of this approach, including all existing FF classification methods, is given in Sections 3.5.1 and 3.5.2.

A second approach aims at the identification of all FSM gates, including FFs and combinational gates of the FSM, within one step. Thus, this second approach combines the identification steps 1) and 2) of the first approach within a single process. One can use partitioning and/or matching algorithms to cluster and/or classify netlist gates into functional modules, including modules that contain the FSMs [e.g. LWS12; Alr+22]. Li et al. [LWS12] assume that a suitable library component and netlist partitioning exist to achieve a successful matching of the FSM gates. The work in [Alr+22] uses a GNN to identify FSM gates by clustering them into one joint group.

Although data path identification and FSM identification have different objectives, the two topics are related and can complement each other [e.g. Mea+16; Li+13]. An identified data word, for example, excludes the corresponding FFs as possible candidates for an FSM, and vice versa.

- **Functionality identification:** In contrast to the identification of specific components, like the data path or FSMs, this step aims to identify the general, high-level functionality of modules or partitions, like adder and multiplier, but also cryptographic algorithms, like the AES. A typical approach is to match the unknown functionality to modules, partitions, subgraphs, or structures with known functionality, which are, for example, stored in a so-called module library. The matching can, for example, be done by comparing the partitions' input-output behavior, e.g. by solving a satisfiability problem on the partitions' Boolean functions [Sub+13; Sub+14] or using binary decision diagrams [Shi+12], by identifying module-specific netlist structures, like the ones for counters [Sub+13; Sub+14], by comparing pattern graphs [LWS12], or by comparing structural netlist characteristics. State-of-the-art matching with structural characteristics usually performs fuzzy matching because due to synthesis or physical RE errors, there will not be a perfect structural equivalent in the module library. For structural comparison, one can, for example, use feature extraction together with classical classification approaches, like *k-nearest neighbor* [Bae+19], or with sophisticated machine learning algorithms, like GNNs [Alr+22]. Other methods combine different techniques, like including functional and structural characteristics into a hash [Bae+19].

- **Input/Output signal identification:** This analysis step is not the most obvious to be included for functional gate-level netlist RE. One could also add it as one of the last steps for physical RE to indicate, for example, what type of memory provides what signals to the netlist. However, especially for library matching approaches, it could be an advantage to identify not only the external inputs and outputs of a design but also the incoming and outcoming signals of separate partitions [e.g. LWS12; Fyr+17]. This target will rather count to functional gate-level netlist RE.

3 Features of Finite State Machines

This chapter first introduces the basics and terminology of FSMs. Next, it analyzes the degrees of freedom of Moore and Mealy machine definitions, which are then used to derive sFF and sFF set properties. These properties are used to develop a systematic categorization of sFF sets and an sFF set definition for human-readable state machines. Finally, sFF identification algorithms are introduced, analyzed, and classified based on their used features and based on the previously derived systematic categorization.

Parts of this chapter have already been pre-published in the author's works [BBS19], [Bru+22a], [Bru+22b], and [Bru+23]. Section 3.1 is composed of the works in [Bru+22a] and [Bru+22b]. Sections 3.2 to 3.4 are based on [Bru+22a], while the part about Strongly Connected Components in Section 3.3 is additionally based on [Bru+23]. Finally, the background sections (Sections 3.5.1 and 3.5.2) are based on the works in [Bru+22a], [Bru+23], and [BBS19] or the author's master thesis [Bru18], while Section 3.5.3 is based on [Bru+23] and Section 3.5.4 is based on [Bru+22a].

3.1 Finite State Machine Basics

The well-known definitions of Moore and Mealy machines are based on the work of Moore [Moo56] and Mealy [Mea55], respectively. In [Moo56], the automaton used by Moore for his gedankenexperiment is defined as a synchronous, finite machine with deterministic behavior. A Moore machine $\mathcal{M}$ can be defined as a 6-tuple

$$\mathcal{M} = (\mathbb{S}, S^*, \mathbb{I}, \mathbb{O}, \delta, \lambda),$$

in which $\mathbb{S} = \{S_m, \ldots, S_1\}$ is the finite set of states, $S^* \in \mathbb{S}$ is the initial state, $\mathbb{I} = \{I_k, \ldots, I_1\}$ is the finite set of inputs, $\mathbb{O}$ is the finite set of outputs, $\delta : \mathbb{S} \times \mathbb{I} \to \mathbb{S}$ is the transition function, and $\lambda : \mathbb{S} \to \mathbb{O}$ is the output function. The mathematical model of the Mealy machine is similar to the model of the Moore machine but with a different output function $\lambda : \mathbb{S} \times \mathbb{I} \to \mathbb{O}$.

An FSM can be represented as a State Transition Graph (STG) or as a State Transition Table (STT). For an STG, the graph nodes represent states, while the graph edges (with labels) represent state transitions (with transition conditions) [Ped13]. Instead, for an STT, each row represents one valid state transition. The row gives the current state, the next state, and the condition to perform the transition from this current to this next state. Usually, the transition conditions are not provided with a Boolean logical format but by assigning binary values to the input variables. Thus, often, multiple rows are required to cover all possible transition conditions from one current state to one next state. Such a set of q transition conditions from a state S_i to a state S_j is defined by the set $\mathbb{E}(S_i, S_j) = \{e_q(S_i, S_j), \ldots, e_1(S_i, S_j)\}$.

To translate the mathematical descriptions of an FSM into hardware netlist structures, the machines are represented by a next state and output logic as well as memory elements [Ped13], as shown in Figure 3.1. The next state and output logic consists of gates out of the set of combinational gates $\mathbb{N}$ and wires. Besides, an FSM state is formed by n state bits, which are held by a set of sFFs, the sFF set $\mathbb{F}^s = \{F_n^s, \ldots, F_1^s\}$ or by a state register in a gate-level netlist. A register is a group of FFs which belong together, e.g. form a variable in the RTL code. The next state is determined by the next state logic, elements of $\mathbb{I}$, and elements of $\mathbb{F}^s$. This

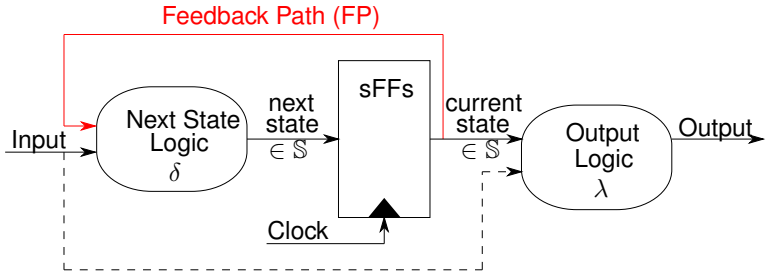


Figure 3.1 Hardware netlist representation of Moore machine and Mealy machine (difference highlighted with dashed line): combinational next state and output logic, state register with FP (highlighted in red)

results in a connection from the state register's output to the state register's input, see the red line in Figure 3.1. We call a connection from an output of a netlist gate, FF or register to its input a Feedback Path (FP). All remaining FFs are called Non State Flip-Flops (nFFs) which form an nFF set $\mathbb{F}^n = \{F_u^n, \ldots, F_1^n\}$. Consequently, the union of both sets, $\mathbb{F}^s \cup \mathbb{F}^n$, form the set of all FFs $\mathbb{F} = \{F_{n+u}, \ldots, F_1\}$. The value of F_j^s, $j = \{n, \ldots, 1\}$ at time t, $f_j^{s,t}$, and of I_l, $l = \{k, \ldots, 1\}$ at time t, i_l^t, is either '0' or '1', so an inversion, e.g. $\overline{f_j^{s,t}}$, flips the value and a state transition can be described as follows:

$$f_n^{s,t} ... f_1^{s,t} \times i_k^t ... i_1^t \rightarrow f_n^{s,t+1} ... f_1^{s,t+1}$$

There exist different strategies to encode FSM states [Ped13]. Two of the best-known ones are one-hot and binary encoding. One-hot encoding assigns one sFF to each state, encodes it to '1' and all other sFFs to '0'. This results in $n = m$ required sFFs. Binary encoding uses the complete range of binary values to represent FSM states, resulting in $n = \lceil \log_2(m) \rceil$ required sFFs.

A Mealy and a Moore machine implementing the same functionality will feature different STGs and thus different netlist representations. For example, transforming a Mealy into a Moore machine usually increases the number of FSM states and transitions [KS10]. Instead, a Moore and a Mealy machine consisting of the same $\mathbb{S}$ and δ will feature equal netlist representation for next state logic and sFFs. Thus, without loss of generality, the following work focuses on the Moore machine.

3.2 Degrees of Freedom of Moore and Mealy Machine Definitions

To reverse engineer an FSM in a netlist, it is important to understand the process of FSM implementation. A hardware designer or a synthesis suite has some degrees of freedom to describe an FSM. By analyzing the properties of a Moore or Mealy machine description, it is possible to derive two degrees of freedom entailed: the connectivity of the combinational logic and the definition of the state set, see Figure 3.2. Parts of the circuitry in the FSM's proximity may be associated with the Moore or Mealy machine or may be considered unrelated to the machine. For instance, if a counter influences an FSM transition, it can either be considered as part of this FSM or as separate logic that is used as input condition for this FSM.

3.2.1 Connectivity of State Flip-Flops

The first degree of freedom is the degree of connectivity that is considered to be required within the combinational logic of a self-contained machine, as well as the properties of these

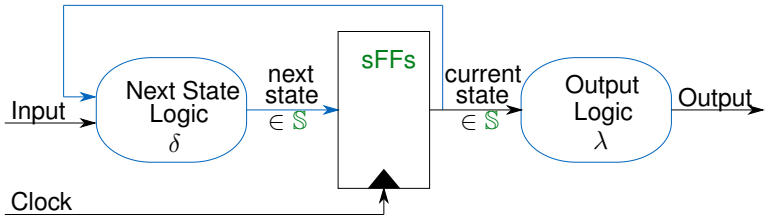


Figure 3.2 Model of a Moore machine with degrees of freedom: connectivity of combinational logic (blue), and definition of state set $\mathbb{S}$ (green)

connections. The blue-colored parts of Figure 3.2, i.e. the next state and output logic, could be considered as two or multiple independent, partly dependent, or dependent parts without violating the Moore machine description in Section 3.1.

The topic of separating FSMs into individual machines is also known as FSM decomposition [Har60]. However, the objective of FSM decomposition, as given in the literature, is the optimization of area and performance [DN88], logic complexity [ADN91] or power [MO98]. In this work, FSM decomposition is analyzed independently of optimization goals in a more general manner. Nevertheless, there are several similar concepts, such as the notion of three different categories of decomposition, called parallel decomposition, cascade decomposition, and general decomposition [DN88]. These correspond to this section's independent, partly dependent, and dependent scenarios. In the field of sequential RE, FSM decomposition was considered by Meade et al. [MZJ16]. In that work, a heuristic decomposition approach for directed graphs is presented to split an extracted FSM into its individual, independent FSMs after the FSM extraction procedure.

The following discussion first concentrates on the connectivity analysis of the next state logic, i.e. the connectivity that is considered to be required between the sFFs of a self-contained machine. The connectivity of the output logic is briefly investigated at the end of this subsection.

To investigate the connectivity of sFFs of an FSM, we separate the hardware netlist representation of an FSM into multiple parts. In the following, $\mathbb{P} = \{\mathbb{F}_1^s, \mathbb{F}_2^s\}$ is defined as a partition of an sFF set $\mathbb{F}^s$ into two separate sFF sets $\mathbb{F}_1^s$ and $\mathbb{F}_2^s$. They fulfill the following conditions

$$\mathbb{F}_{1,2}^s \subset \mathbb{F}^s; \quad \mathbb{F}_1^s \cup \mathbb{F}_2^s = \mathbb{F}^s; \quad \mathbb{F}_1^s \cap \mathbb{F}_2^s = \emptyset,$$

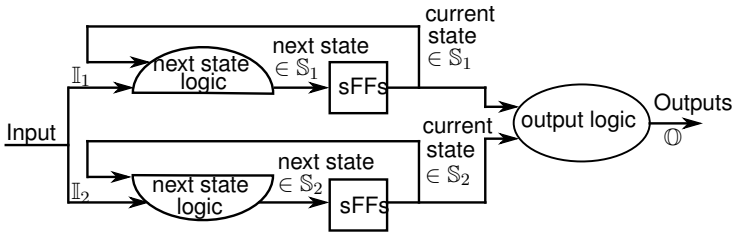
Thus, an sFF $F^s \in \mathbb{F}^s$ can only be part of one of the two subsets of the partition $\mathbb{P}$. Without loss of generality, we only consider partitions with two subsets because every partition that contains more than two subsets can also be represented by a partition that contains only two subsets.

In the following, the three connectivity definitions, independent, partly dependent, and dependent, are introduced and discussed. For this, the definition of a netlist path function $P(a, b)$ in Definition 2 is used.

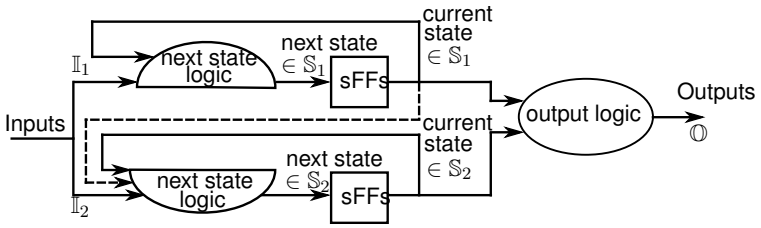
Definition 3 (Independent parts) *An FSM implementation can be separated into independent parts if its sFF set can be separated such that separate FPs of the corresponding sFF sets and no further interconnection between the two separated parts exist.*

This means that a partition $\mathbb{P}$ of the sFF set $\mathbb{F}^s$ into independent parts $\mathbb{F}_{1,2}^s$ exists, in which each sFF $F^s \in \mathbb{F}_{1,2}^s$ still has an FP, and which has the property that

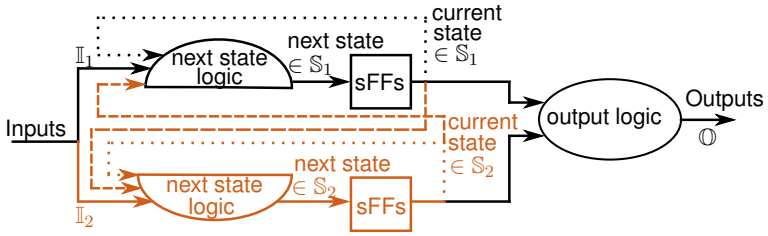
$$\forall F_1^s \in \mathbb{F}_1^s, \forall F_2^s \in \mathbb{F}_2^s : \\ \neg P(F_1^s, F_2^s) \wedge \neg P(F_2^s, F_1^s).$$



(a) Two independent parts based on Definition 3



(b) Two partly independent parts based on Definition 4



(c) Two dependent parts based on Definition 5.

Figure 3.3 Model of a Moore machine with independent, partly dependent and dependent parts of next state logic

Figure 3.3a shows an example of a split into two independent parts. The example could still be interpreted as one joint machine with $\mathbb{S} = \mathbb{S}_1 \cup \mathbb{S}_2$ and $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$. However, the sFF set can also be separated into two sFF sets according to Definition 3. So, the more reasonable interpretation are two machines, namely $\mathcal{M}_1 = (\mathbb{S}_1, S_1^*, \mathbb{I}_1 \cup \mathbb{S}_2, \mathbb{O}, \delta_1, \lambda)$ and $\mathcal{M}_2 = (\mathbb{S}_2, S_2^*, \mathbb{I}_2 \cup \mathbb{S}_1, \mathbb{O}, \delta_2, \lambda)$, with $S^*_{\{1,2\}} \in \mathbb{S}_{\{1,2\}}$, $\delta_{\{1,2\}} : \mathbb{S}_{\{1,2\}} \times \mathbb{I}_{\{1,2\}} \to \mathbb{S}_{\{1,2\}}$ and $\lambda : \mathbb{S}_1 \times \mathbb{S}_2 \to \mathbb{O}$, respectively. The definition of the output function λ is the same for both machines, but the interpretation of the sets $\mathbb{S}_1$ and $\mathbb{S}_2$ differs, depending on which machine is considered. For $\mathcal{M}_1$, the set $\mathbb{S}_1$ represents the state set, while $\mathbb{S}_2$ represents a part of the input set and vice versa. Consequently, the joint machine, which was modeled as a Moore machine, can be interpreted as two machines following the Mealy machine model, and the input alphabets for the two machines $\mathcal{M}_1$ and $\mathcal{M}_2$ are extended by the sets $\mathbb{S}_2$ and $\mathbb{S}_1$, respectively.

Definition 4 (Partly Independent parts) *An FSM implementation can be partly separated if its sFF set cannot be separated according to Definition 3 but can be separated such that separate FPs of the corresponding sFF sets and a connection in only one direction between the two parts exist.*

This means that a partition $\mathbb{P}$ of the sFF set $\mathbb{F}^s$ into partly independent parts $\mathbb{F}^s_{1,2}$ exists, in which each sFF $F^s \in \mathbb{F}^s_{1,2}$ still has an FP, and which has the property that

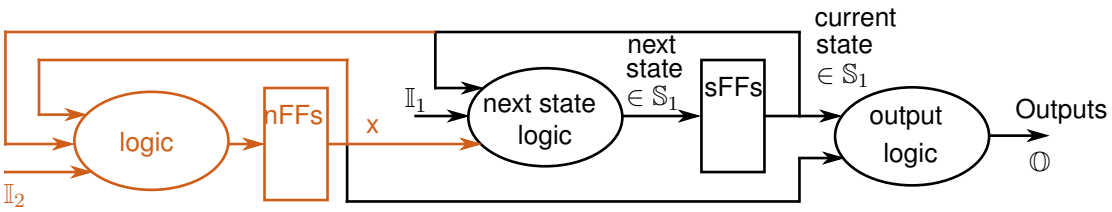


Figure 3.4 Model of a Moore machine which redefines state set $\mathbb{S}$ compared to the model in Figure 3.3c. Only black FFs are interpreted as sFFs.

$$\begin{aligned}&\exists\, \mathbb{F}_i^s :\\ &\quad \forall F_i^s \in \mathbb{F}_i^s, \forall F_j^s \in \mathbb{F}_j^s :\\ &\qquad\qquad P(F_i^s, F_j^s) \wedge \neg P(F_j^s, F_i^s);\\ &\quad i, j \in \{1, 2\}; \quad i \neq j.\end{aligned}$$

Figure 3.3b shows an example of a split into two partly independent parts. The example can again be interpreted as one joint machine or as two machines. The joint machine interpretation has the same state and input set as the scenario before, $\mathbb{S} = \mathbb{S}_1 \cup \mathbb{S}_2$ and $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$. However, the sFF set can also be separated into two sFF sets according to Definition 4. So, if the example model in Figure 3.3b is interpreted as two machines, the resulting machine definitions are the same as the ones according to Definition 3, except for the next state function of the second machine, $\delta_2 : \mathbb{S}_2 \times (\mathbb{I}_2 \cup \mathbb{S}_1) \to \mathbb{S}_2$, which now also uses the extended input alphabet.

Definition 5 (Dependent parts) *An FSM implementation contains only dependent parts if its sFF set cannot be separated according to Definitions 3 or 4 but can be separated only such that a connection in both directions between the corresponding sFF sets exists.*

This means that a partition $\mathbb{P}$ of the sFF set $\mathbb{F}^s$ into dependent parts $\mathbb{F}_{1,2}^s$ exists, in which each sFF $F^s \in \mathbb{F}_{1,2}^s$ still has an FP, and which has the property that

$$\begin{aligned}&\forall F_1^s \in \mathbb{F}_1^s, \forall F_2^s \in \mathbb{F}_2^s :\\ &\qquad\qquad P(F_1^s, F_2^s) \wedge P(F_2^s, F_1^s).\end{aligned}$$

Figure 3.3c shows an example of a split into two dependent parts (according to Definition 5). Again, this presentation can be interpreted as one joint machine or two separate machines. However, in this case, the interpretation of a joint machine is more reasonable, using again $\mathbb{S} = \mathbb{S}_1 \cup \mathbb{S}_2$ as state set and $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$ as input set. If the model in Figure 3.3c represents two machines, the resulting machine definitions are the same as the ones according to Definition 4, except for the next state function of the first machine, $\delta_1 : \mathbb{S}_1 \times (\mathbb{I}_1 \cup \mathbb{S}_2) \to \mathbb{S}_1$. Definition 5 does not require the FPs for the two sFF sets in Figure 3.3c (dotted lines) to exist because the additional connections (dashed orange lines) still ensure an FP for each sFF.

Similar to the next state logic, the output logic can also be analyzed concerning its degree of connectivity. As a result, some of the Mealy machines of the separate automata interpretations can be changed back into Moore machines if a particular separation of the output logic is possible. This also enables undoing the input alphabet extensions performed in the steps above.

3.2.2 Definition of the State Set and its Representation in Hardware

The Moore and Mealy machine descriptions allow a second degree of freedom besides the connectivity of sFFs in the previous section. The second degree of freedom is the definition of the state set $\mathbb{S}$, see the green colored part of Figure 3.2. A state set contains several states defined as part of the FSM. Those states are represented by state bits, which are translated to sFFs in the gate-level netlist. Thus, each $\mathbb{S}$ has an associated sFF set and each state in $\mathbb{S}$ is represented by a specific logical state of the sFFs in the sFF set [1]. The associated sFFs are also colored in green in Figure 3.2. The Moore and Mealy machine definitions in Section 3.1 do not further define $\mathbb{S}$. Thus, the resulting degree of freedom is the content of the state set $\mathbb{S}$ for a machine $\mathcal{M}$, and consequently the content and construction of the corresponding sFF set.

The relevance of this discussion is shown when comparing the representations in Figure 3.3c and Figure 3.4. In Figure 3.3c, all FFs are interpreted as sFFs, i.e. $\mathbb{F}^s = \mathbb{F}^s_1 \cup \mathbb{F}^s_2$ appropriate for the state set $\mathbb{S} = \mathbb{S}_1 \cup \mathbb{S}_2$. We could, however, exclude, for example, the orange-colored FFs from the sFF set. In this case, the model can be redrawn, resulting in the following state machine: $\mathcal{M} = (\mathbb{S}_1, S^*_1, \mathbb{I}_1 \cup x, \mathbb{O}, \delta, \lambda)$, with $S^*_1 \in \mathbb{S}_1$, $\delta : \mathbb{S}_1 \times (\mathbb{I}_1 \cup x) \to \mathbb{S}_1$ and $\lambda : \mathbb{S}_1 \times x \to \mathbb{O}$, see Figure 3.4.

3.3 Derivation of State Flip-Flop and State Flip-Flop Set Properties

In this section, we derive two sFF properties and one sFF set property using the investigated degrees of freedom for Moore and Mealy machines of Section 3.2.

3.3.1 State Flip-Flop Property Definitions

The two sFF property definitions are derived from the connectivity analysis results of Section 3.2.1. The first property definition is also called *connectivity*, which now describes the existence of paths between sFFs without considering their composition. The composition is described by the second property definition, called *path*. Thus, to define the sFF property *connectivity*, the definition for the netlist path function (Definition 2) is used, whereas to define the sFF property *path*, the definition for the netlist path (Definition 1) is used.

Connectivity

The more reasonable interpretations of FSM structures according to the definitions in Section 3.2.1, namely separated machines for independent parts and a joint machine for dependent parts, result in two possible connectivity properties to describe which sFFs belong to the same machine. For both, an sFF of machine $\mathcal{M}$ has to possess an FP, but the paths within the set of all n sFFs in the sFF set $\mathbb{F}^s_{\mathcal{M}} = \{F^s_1, \ldots, F^s_n\}$ of machine $\mathcal{M}$, called cross-FF-influence, vary. An sFF belongs to machine $\mathcal{M}$ in the strong sense if it possesses an FP and influences and is influenced by all other sFFs of machine $\mathcal{M}$. Such sFFs will be called strong sFFs.

Definition 6 (Strong sFF) *The F^s_i is a strong sFF of machine $\mathcal{M}$ if*

$$\underbrace{P(F^s_i, F^s_i)}_{\text{FP}} \wedge \underbrace{\bigwedge_{k \in \{1,\ldots,n\} \setminus \{i\}} \left(P(F^s_i, F^s_k) \wedge P(F^s_k, F^s_i)\right)}_{\text{cross-FF-influence}}$$

[1] Note that this is a unidirectional relationship. Not every logical state of the sFF set has to represent a state in $\mathbb{S}$. This depends on the state encoding, the state bit count, or the synthesis optimizations.

A sFF belongs to machine $\mathcal{M}$ in the weak sense if it possesses an FP and either influences or is influenced by each of the other sFFs of the machine $\mathcal{M}$, or both. Such sFFs will be called weak sFFs.

Definition 7 (Weak sFF) *The F_i^s is a weak sFF of machine $\mathcal{M}$ if*

$$\underbrace{P(F_i^s, F_i^s)}_{FP} \wedge \underbrace{\bigwedge_{k \in \{1,\ldots,n\} \setminus \{i\}} (P(F_i^s, F_k^s) \vee P(F_k^s, F_i^s))}_{\text{cross-FF-influence}}$$

Definition 6 and 7 show that each strong sFF also satisfies the definition of a weak sFF, but not vice versa.

According to Definition 3 (independent parts), the sFFs, which are part of the joint machine interpretation, are neither weak nor strong sFFs of the joint machine. According to Definition 4 (partly independent parts), the sFFs, which are part of the joint machine interpretation, are weak, but no strong sFFs of the joint machine. According to Definition 5 (dependent parts), the sFFs, which are part of the joint machine interpretation, are weak and strong sFFs of the joint machine. The decision, if a given machine can be interpreted as a single joint machine, can thus be formulated with respect to the allowed connectivity property of sFFs. For example, if strong sFFs are required, only the machine according to Definition 5 can be interpreted as a single joint machine. If also weak sFFs are allowed, the joint interpretation is possible for the machine according to Definition 4, as well.

Path

The sFF property *connectivity* only considers the existence of paths between sFFs. The sFF property *path* now defines the composition of these paths, including the composition of the FPs. We define three path classes: generic, FSM, and combinational.

Definition 8 (Generic Path Class) *A generic path $p(F_i^s, F_j^s)$ can be composed of gates g that are sFFs, nFFs or combinational:*

$$\begin{aligned} &\forall g \in p : \\ &\quad g \in \mathbb{F}^s \vee g \in \mathbb{F}^n \vee g \in \mathbb{N}, \end{aligned}$$

in which $\mathbb{F}^s$ is the sFF set of the design, $\mathbb{F}^n$ is the nFF set of the design and $\mathbb{N}$ is the set of combinational gates of the design.

Definition 9 (FSM Path Class) *An FSM path $p(F_i^s, F_j^s)$ can be composed of gates g that are sFFs or combinational:*

$$\begin{aligned} &\forall g \in p : \\ &\quad g \in \mathbb{F}^s \vee g \in \mathbb{N}. \end{aligned}$$

Definition 10 (Combinational Path Class) *A combinational path $p(F_i^s, F_j^s)$ can solely be composed of gates g that are combinational:*

$$\begin{aligned} &\forall g \in p : \\ &\quad g \in \mathbb{N}. \end{aligned}$$

Note that the set of all combinational paths in a design is a subset of all FSM paths, which likewise is a subset of all generic paths. Paths of at least generic/FSM/combinational class will be called *minimum* generic/FSM/combinational paths. A minimum generic path is defined as a path of generic, or even FSM or even combinational class, a minimum FSM path is defined as a path of FSM or even combinational class, and a minimum combinational path is defined as a path of combinational class only.

3.3.2 State Flip-Flop Set Property Definition

The sFF set property *condition of membership* can be derived by considering the state set discussion in Section 3.2.2. The condition of membership describes the condition which holds for constructing sFF sets out of a number of sFFs.

Definition 11 (Necessary condition of membership) *A sFF set $\mathbb{F}^s$ has the necessary condition of membership and is called a $\mathbb{F}^s_N$, if a {strong, weak} connectivity property of an sFF is a necessary condition of membership for the sFF set of machine $\mathcal{M}$:*

$$F \in \mathbb{F}^s_N \rightarrow F \textit{ is \{strong, weak\} sFF},$$

i.e. if an FF F is member of the $\mathbb{F}^s_N$, this implies that it is a strong or weak sFF of machine $\mathcal{M}$.

Definition 12 (Sufficient condition of membership) *A sFF set $\mathbb{F}^s$ has the sufficient condition of membership and is called a $\mathbb{F}^s_S$, if a {strong, weak} connectivity property of an sFF is a sufficient condition of membership for the sFF set of machine $\mathcal{M}$:*

$$F \in \mathbb{F}^s_S \leftarrow F \textit{ is \{strong, weak\} sFF},$$

i.e. if an FF F is a strong or weak sFF of machine $\mathcal{M}$, this implies that this FF is defined to be a member of the $\mathbb{F}^s_S$.

The discussion about a varying sFF set is only possible for a $\mathbb{F}^s_N$, but not for a $\mathbb{F}^s_S$.

In graph theory, the literature defines another set of nodes, called SCC, and algorithms to identify it [e.g. Tar71]. Even though this set is not designed for sFFs, it is often used in relation to sFFs.

Definition 13 (Strongly Connected Component) *The literature defines an SCC as a set of connected nodes in a graph with the following properties:*

1. *there is a path in the graph from every node to every other node in the set,*
2. *every node which satisfies SCC-property 1 is part of the set.*

When mapping gates to nodes and wires to edges, an SCC thus is a set of strongly connected gates. When further ignoring all combinational gates by replacing them with wires, an SCC can also be a set of strongly connected FFs. There exists a difference between an sFF set which fulfills strong connectivity as defined in Definition 6 and an sFF set which forms an SCC. For both, SCC-property 1 holds. However, only for an SCC also SCC-property 2 holds because Definition 6 does not include conditions about the completeness of the set. Though this is achieved by the sufficient condition of membership in Definition 12. Consequently, the sFFs which are part of an sFF set which forms an SCC are strong sFFs with minimum FSM path class and a sufficient condition of membership.

The authors in [Tar71] developed Tarjan's algorithm, an efficient algorithm to identify SCCs. Using Tarjan's algorithm, one can identify all SCCs within a netlist. For the following work, we label an SCC specifically, namely *FSM SCC*, if it contains all or the majority of sFFs of the original FSM. Depending on the properties that the sFF set of the original FSM fulfills and due to SCC-property 2, the FSM SCC might also contain other FFs in addition to the sFFs of the original FSM. All other SCCs which contain at least two FFs are labeled as data SCCs.

3.4 Usage of Derived State Flip-Flop and State Flip-Flop Set Properties

The sFF and sFF set property definitions are now combined to design a systematic categorization of sFF sets and to derive a novel sFF set definition for a human-readable state machine.

3.4.1 Systematic Categorization of State Flip-Flop Sets

Each connectivity property, weak and strong, can be systematically combined with each path class, generic, FSM, and combinational, to form six new property combinations. In the rest of the thesis, these property combinations will be called (sFF set property) categories. All possible sFF property combinations are shown in Figure 3.5. Combining the sFF properties *connectivity* and *path* implies specifying a minimum path class for all paths in the corresponding connectivity Definitions 6 and 7 (strong and weak sFFs). The novel categorization can be used to categorize sFFs and sFF sets. An sFF is sorted in the category with weak/strong connectivity and generic/FSM/combinational path class if it satisfies weak/strong connectivity and minimum generic/FSM/combinational paths. On the other hand, an sFF set is sorted in the category with weak/strong connectivity and generic/FSM/combinational path class if all its sFFs are weak/strong sFFs with minimum generic/FSM/combinational paths. The following work primarily categorizes sFF sets, including the output sFF sets of sFF identification methods (see Section 3.5.4) or the sFF sets of the designs' FSMs (see Section 4.2.2).

For the majority of the resulting categories, all associated sFF sets clearly satisfy the Moore or Mealy machine definition with its degrees of freedom, see Section 3.2. Only sFF sets with weak or strong sFFs and a generic, and not FSM or combinational FP do not clearly satisfy the Moore and Mealy definitions. The Moore or Mealy machine definitions seem to allow only sFFs with combinational or FSM FPs. However, the synthesis might optimize specific FPs, resulting in generic FPs, see the example in Section 4.2.2. There, the state encoding and special state transitions allow such a synthesis optimization. Thus, also sFF sets which contain weak or strong sFFs with generic FPs are considered to satisfy Moore and Mealy machine definitions.

In a given sFF set, there might also exist FFs which are neither weak nor strong sFFs. sFF sets which contain such FFs are sorted in none of the defined categories, visualized by the dark green colored area in Figure 3.5. Examples for such FFs are the sFFs of a joint machine interpretation according to Definition 3 (independent parts), or FFs without an FP. The most reasonable interpretations of such sFF sets are that their elements belong to multiple separate FSMs or no FSM. All other sFF sets are assumed to represent one joint state machine, see the light green colored area in Figure 3.5.

The sFF set property *condition of membership* defines if all (sufficient condition of membership) or only some (necessary condition of membership) of the FFs which fulfill the required sFF properties of the category are part of this particular sFF set.

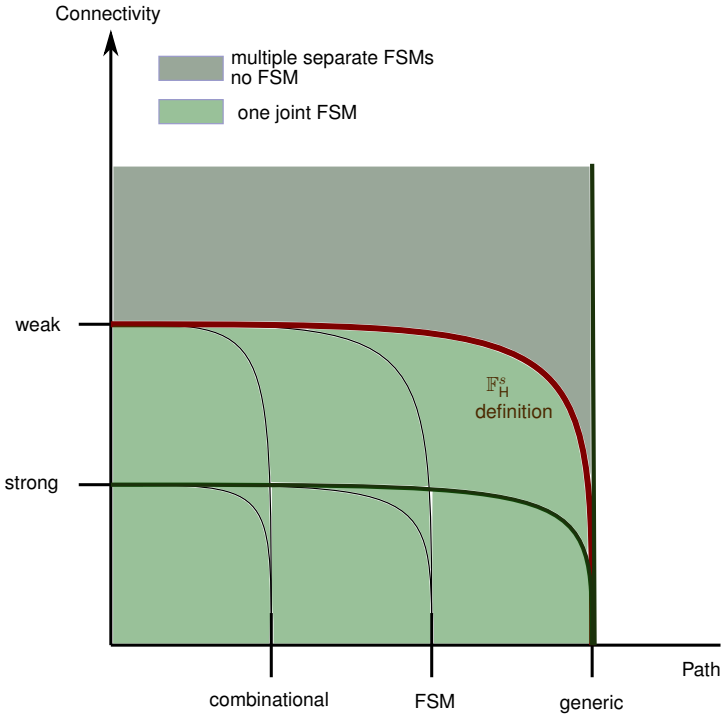


Figure 3.5 Overview of possible sFF set categories. Combining each connectivity, weak and strong, with each path class, generic, FSM, and combinational, results in six categories. All sFF sets which are at least part of the category weak connectivity and generic path class are interpreted as one joint FSM (see light green colored area), while all other sFF sets are interpreted as multiple separate FSMs or no FSM (see dark green colored area). The boundary between both interpretations is additionally illustrated by a red line and labeled by "$\mathbb{F}_{\mathrm{H}}^{s}$ definition". This label marks the newly presented sFF set definition in Section 3.4.2. Note that the categories, like the properties, are not mutually exclusive. For example, an sFF with strong connectivity and FSM path class also fulfills the properties weak connectivity and generic path class.

3.4.2 State Flip-Flop Set Definition for a Human-Readable State Machine

Section 3.5 will show that sFF identification strategies gear towards sFFs with different sFF and sFF set properties. As stated in the introduction, RE wants to increase the understanding of the design. Thus, we want to improve sequential RE efforts towards a human-readable state machine extraction. This objective of sequential RE must be carried into the sFF identification strategies and into the sFF set definition which represents the optimum result for an sFF set. This allows to suggest a possible definition of an sFF set using the derived sFF and sFF set properties. The definition is based on the following three assumptions:

1. The implementation of the FSM states in the original FSM source code uses FFs which are weak or strong sFFs with minimum generic paths.

2. The sFF set obtained by collecting all FFs of assumption 1 satisfies the necessary condition of membership. A sufficient condition of membership is not required.

3. There is only one FSM in the design.

The last assumption simplifies the further discussion. However, if more than one FSM exists in a design, the following definition can also be applied separately.

Definition 14 (sFF set definition for a human-readable state machine) *The sFF set $\mathbb{F}^s_H$ contains all sFFs which are used to define the machine states in the design description of the source code, in consideration of assumptions 1-3.*

This definition is strongly related to the design description of the machine source code, which would be the optimum result of a human-readable extraction of the FSM. Hence, this definition is the sFF set definition for a human-readable state machine.

Based on the three assumptions above, $\mathbb{F}^s_{\mathrm{H}}$ is sorted in the category with weak connectivity and generic path class and, therefore, encloses all possible categories for a single joint machine, see the red line in Figure 3.5. Typical $\mathbb{F}^s_{\mathrm{H}}$ sFF sets fulfill the necessary condition of membership but not the sufficient condition of membership.

3.5 State Flip-Flop Identification Methods

The previous sections derived sFF and sFF set properties and thus categories by analyzing the gate-level netlist structure of FSMs and their sFFs. This section analyzes and categorizes existing sFF identification techniques and explores the features they use to differentiate sFFs from nFFs. The following section first explains the role of sFF identification in FSM extraction and gives an overview of state-of-the-art sFF identification methods. Next, it categorizes the methods using the two different approaches: 1) based on all used features and 2) based on the categories defined in Section 3.4.1.

3.5.1 The Role of State Flip-Flop Identification in Finite State Machine Extraction

FSM extraction is an essential part of sequential hardware netlist RE, see Section 2.3. A correct FSM identification and extraction mostly relies on a correct sFF identification in the gate-level netlist, as this is the first and most error-prone step. Once the correct sFFs are identified in the gate-level netlist, sequential RE identifies all other, combinational gates of the FSM and extracts the FSM functionality [e.g. McE01; MZJ16].

The patent in [McE01] suggests determining a minimum extraction region based on the identified sFFs. The minimum extraction region consists of combinational gates which reach one or multiple inputs of sFFs and are reached by one or multiple outputs of sFFs, which means combinational gates which are part of the sFFs' FPs. Next, the minimum extraction region is extended by adding further combinational gates to identify the minimum set of input variables that drive the minimum extraction region. Thus, for each sFF input, there exists a corresponding identified FSM netlist structure. This netlist structure can be translated to a Boolean function. For this, the sFF input is interpreted as the output of the Boolean function, and the input signals and sFF outputs of the netlist structure are interpreted as inputs of the Boolean function. The FSM functionality can then be extracted based on the identified Boolean functions.

The works in [MZJ16; MZJ17] introduce an FSM extraction algorithm REFSM. It prunes the gate-level netlist by removing FFs which are assumed to be less important, i.e. FFs which are farther than a predefined distance from the given sFFs. The algorithm is repeated with a reduced distance if the pruned graph is not feasible. REFSM extracts the FSM functionality based on Boolean functions, which are generated from the pruned graph. Similar to what is suggested in [McE01], the algorithm applies current state values to the FF output signals and arbitrary values to all remaining Boolean function inputs. The derived outputs represent the next states. For the first run, the algorithm applies initial values, ideally the reset state, to the

FF output signals and each possible input pattern combination to the remaining inputs. The reset state can be defined as a state that is reachable from every state in the FSM [McE01]. As a result, one gets a list with all possible next states, together with the corresponding transition conditions, for the reset state. The process is repeated for a newly identified next state until all derived states are explored.

In [Mea+19], REFSM is improved to also handle uncommon structures, like an XOR tree. The work in [Fyr+18] determines the combinational gates of an FSM candidate similar to [McE01] and extends the work in [MZJ16]. It additionally allows the detection of counters to prevent them from being wrongly classified as FSMs. Recently, Geist et al. [Gei+23] present RECUT and REFSM-SAT. RECUT pre-processes the gate-level netlist which is applied to the FSM extraction algorithm. It determines the input cones of the provided sFFs by a breadth-first search, starting from the sFFs' inputs until an FF output or a primary input is reached. On the other hand, REFSM-SAT is more performant than REFSM but cannot provide the transition conditions for the resulting state transitions.

3.5.2 Overview of State-Of-The-Art Techniques

The previous subsection shows that as long as the correct sFFs are identified, mainly the determination and evaluation of Boolean functions remain to extract the correct FSM. Identifying the correct sFFs from a reverse-engineered gate-level netlist, however, is a challenging task for which currently only heuristic approaches exist [Azr+21]. Consequently, the success of retrieving a correct FSM can be measured by the success of identifying the correct sFFs.

In the following, we will introduce existing methods to identify sFFs from a reverse-engineered gate-level netlist. This excludes methods that assume knowledge about the netlist beyond the knowledge one can achieve from RE a product, like the RTL naming of variable signals in [Kib+22] or an entire synthesis report in [Nah+16]. Such knowledge would significantly ease the sFF identification; thus, such methods are not further considered.

Initial sFF identification methods: To the best of our knowledge, the methods [McE01] and [Shi+10] identify sFFs to be FFs with a combinational FP [McE01] or to be FFs with a combinational FP and an influence on control signals [Shi+10]. Shi et al. [Shi+10] additionally group FFs based on enable signals and SCCs to identify suitable registers.

RELIC: The authors of RELIC [Mea+16] classify FFs into sFFs and nFFs by determining the dissimilarity of the FFs' input cones. The classification is based on the following assumption: *The input cone structures of sFFs are less similar to the input cone structures of other FFs than is the case for the input cone structures of nFFs.*

Many nFFs form data words, see Section 2.3. Thus, in a design that, for example, carries out a 64-bit multiplication, the 64 nFFs of the related data word have a highly similar input structure to implement this 64-bit multiplication. Instead, the control logic, particularly the FSM, has no similar sFF input cone structures, as each state transition is application-specific.

RELIC calculates similarity scores, which are based on a pre-processed graph and represent an estimation of the input cone structure similarity between two FFs. The final FF classification is done based on the calculated similarity scores. Thus, FFs with less similar input cones are classified as sFFs. In particular, as shown in Figure 3.6, initially, RELIC pre-processes a gate-level netlist to guarantee that a similar functionality is represented by a similar gate structure. For this, RELIC converts the original netlist to a netlist with solely AND, OR, and Inverter gates and merges wires to avoid undesired gate-level discrepancies. To verify the

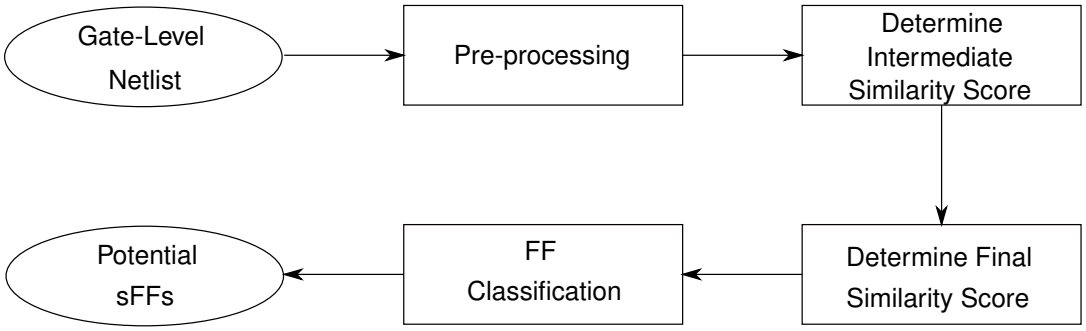


Figure 3.6 Flow diagram of RELIC [Mea+16] ©2019 IEEE [BBS19]

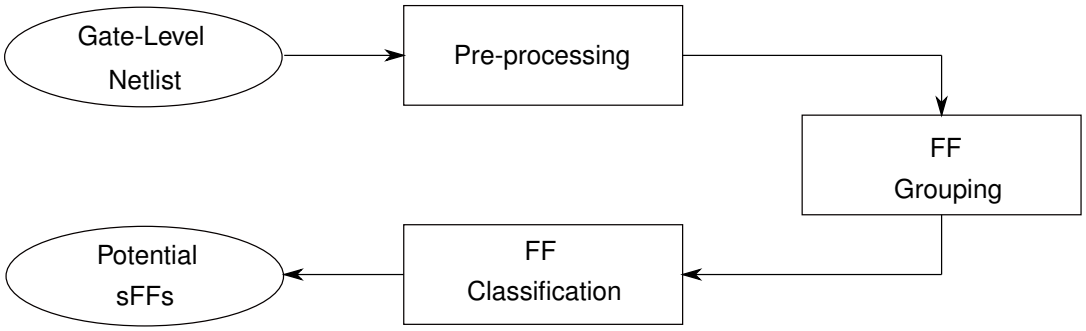


Figure 3.7 Flow diagram of fastRELIC [BBS19] ©2019 IEEE [BBS19]

inverse similarity, one can use a modified merged graph, which is constructed by replacing all AND\OR gates with OR\AND gates and inverting the inputs.

Next, RELIC calculates an intermediate similarity score for each possible FF pair. For this, it recursively compares the gate types of all possible predecessor combinations of an FF pair. It adds an edge to a bipartite graph—which contains the current predecessor gates as nodes—if the gate type comparison reveals a high enough value (threshold $T1$). To avoid an endless recursion of the gate type comparison, a maximum depth d is introduced. If the maximum graph exploration depth d is reached, the algorithm returns the ratio of the minimum and maximum actual predecessors number. Finally, the normalized maximum matching of the resulting bipartite graph determines the intermediate similarity score for the investigated FF pair. If the intermediate similarity score exceeds a predefined threshold $T2$, the counters of the two investigated FFs are increased by one.

If the algorithm calculated an intermediate similarity score for all possible FF pairs, the resulting counters represent the final similarity scores. The FFs are then classified according to the determined similarity scores. A FF is classified as sFF if its counter value does not exceed a predefined threshold $T3$ and classified as nFF otherwise. Based on the work in [Mea+16], several improvements and enhancements were developed, see the following paragraphs for “REWIND”, “fastRELIC”, and “RELIC-FUN”.

fastRELIC: Parallel to the method REWIND [Mea+18], an improvement of RELIC, called fastRELIC, is developed in [BBS19]. fastRELIC is similar to RELIC but reduces the number of required intermediate similarity score calculations. It no longer calculates the intermediate similarity score for all possible FF pairs, but only for a set of special grouped FFs, see Figure 3.7. The intermediate similarity score calculation itself is not changed. Also, similar to RELIC, fastRELIC uses a gate-level netlist and a pre-processing step. As for REWIND, fastRELIC also uses the inverse similarity as an equivalent measure, meaning that for each FF pair, two

intermediate similarity scores are calculated - one with the original merged graphs and one with inverse and the original merged graph. It chooses the maximum of both as the intermediate similarity score for further processing. Additionally, it ignores any inverter gates if they are direct input gates for FFs. Considering the similarity and the inverse similarity as equally relevant prevents the problem of defining two structures that only differ in an inverse FF input as highly different.

The novel grouping algorithm can be denoted as a heuristic, greedy clustering algorithm but is designed for this specific problem. The result is FF groups, which represent the similarity of FFs, and single FFs, which belong to no group. FFs which belong to the same group are assumed to be more similar to each other than to FFs of other groups. The grouping is based on the following assumption:
For an FF F_1 which does not yet belong to a group and an FF F_2 which belongs to a group A: If the intermediate similarity score of F_1 to F_2 is higher than the threshold $T2$, the intermediate similarity score of F_1 to all other members of group A is also higher than the threshold $T2$.

The following describes the process of fastRELIC in detail. Initially, the algorithm chooses a random start FF F_s. Next, it calculates the intermediate similarity score between F_s and all other FFs of the netlist. All FFs with an intermediate similarity score greater than the threshold $T2$ form a group with F_s. All other FFs have to be further investigated. Assuming that FFs with high intermediate similarity scores are more likely to belong to a group than single FFs, they are checked first to reduce the amount of required similarity score calculations. Thus, the algorithm picks the FF which is not yet assigned and which has the highest intermediate similarity score and calculates its similarity to an arbitrary member for every existing group. Suppose the FF cannot be assigned to a group (intermediate similarity score $\leqslant T2$ for all checked groups). In that case, the algorithm calculates the similarity to all remaining, not yet assigned FFs, starting with the FF with the highest intermediate similarity score. If still the FF cannot be assigned to another FF (intermediate similarity score $\leqslant T2$ for all checked FFs), the FF is defined as single FF. This process is repeated until all FFs form a group or are defined as single FFs. All FFs in a resulting group are assumed to be similar. Thus, we interpret the (group sizes -1) to be equal to the RELIC's counter values (final similarity scores). So, the final classification step of fastRELIC is similar to that of RELIC. It classifies all FFs which belong to groups with a group size $\leqslant (T3{+}1)$ as sFFs. The group size of a single FF is defined as one. This results in the potential sFFs and also in the potential nFF sets. Overall, the thresholds for fastRELIC are similar to the ones of RELIC: $T1$, $T2$, d, and $T3{+}1$.

Additionally, fastRELIC extends the similarity-based sFF identification by considering FPs, like it was already done in the previous sFF identification techniques. It classifies only those potential sFFs as sFFs which additionally have a minimum generic FP. Assuming that every sFF must have a minimum generic FP, see Section 3.3.1, the extension does not add sFFs which were wrongly classified as nFF, but can remove nFFs which were wrongly classified as sFFs. Different from [Mea+18], where the FPs are only considered for comparison reasons, the work in [BBS19] exploits the advantage of such an extension.

REWIND: Parallel to the work in [BBS19], an improvement of RELIC is developed in [Mea+18]. Similar to [BBS19], REWIND decreases the complexity of RELIC by reducing the number of required intermediate similarity score calculations with a grouping approach. However, the implementation of the grouping approach differs. REWIND compares a random, yet ungrouped FF F_r to all remaining, ungrouped FFs. Each FF which achieves a higher similarity score than a threshold *t* is grouped together with F_r. Next, the process is repeated for the next random, yet ungrouped FF until all nodes are grouped. An implementation of REWIND is provided in the netlist analysis toolset NetA [Mea+19; MZJ19] under the name *relic_o*.

RELIC-FUN: Geist et al. [Gei+20] present RELIC-FUN, which replaces the structural comparison of RELIC with a functional comparison. Like RELIC, RELIC-FUN also determines the similarity of FFs' input cones. However, in contrast to RELIC, it uses predominantly functional instead of structural features of the netlist. It first identifies an input cone with minimum k inputs for each FF. Next, the algorithm tries to find input cones that provide the same functionality. Thus, it compares a yet ungrouped input cone with one input cone of every existing group. A comparison is only possible if the input cones have the same number of inputs. Two input cones are defined to be similar if one input permutation exists which results in equal outputs of both input cones for all possible input combinations. Finally, similar to fastRELIC and REWIND, the final group sizes indicate if the FF is an sFF or nFF, i.e. FFs of smaller groups are classified as sFFs.

Note that to the best of our knowledge, the work in [Gei+20] as the first defines three groups of FF classification, i.e. high, medium, and low quality FFs, instead of the well known two groups, sFFs and nFFs.

RELIC-Tarjan: The authors of the netlist analysis toolset NetA [Mea+19; MZJ19] suggest an sFF identification approach, called RELIC-Tarjan in the following. For that, they suggest to combine the following NetA tools: *redpen*, *tjscc*, and *relic*.

The tool *relic* extends the method RELIC with Principal Component Analysis (PCA) and structural features, resulting in a Z-Score value for each FF signal. The higher the Z-Score value, the more likely it is that the FF represents an sFF. For the suggested combination, the Z-scores of all FF output signals are calculated with *relic* to identify the most likely sFF, i.e. the signal with the highest Z-Score value. This potential sFF is then used to identify the remaining sFFs.

The NetA tools *redpen* and *tjscc*, an implementation of Tarjan's algorithm, determine all SCCs within the FFs of the netlist. The output of *tjscc* is groups of one or multiple FF signal(s) which form an SCC or single FF signals. The FF signal of a single-element group either forms a self-contained SCC or is not part of any other valid SCC.

Finally, the authors of NetA suggest selecting the SCC which contains the previously determined, most likely sFF. All FF signals which are part of this selected SCC are then interpreted as sFF signals, resulting in an sFF set. We assume that a selected, single-element group is only accepted if it is a self-contained SCC because we assume an sFF to have a minimum generic FP.

Topological analysis: The method in [Fyr+18] combines different approaches of previous methods and adds new techniques. It applies a topological analysis which first groups FFs based on FF types and enable, clock, and reset signals. Next, the algorithm splits these groups of FFs based on existing SCCs in the netlist. FFs which are part of SCCs with a node size smaller than three are removed. It also removes all FFs which do not have a combinational FP, which are the only FF in the SCC, or which do not have a sufficient influence and dependency relation on each other. The authors define that an FF F_1 has a sufficient influence and dependency relation if there are minimum two FFs in the group, both of which are as follows: there is a combinational path from and to F_1. Finally, it removes sFF groups if one of its sFFs does not have enough effect on the control signals.

ReIGNN: A recent method in [CYN21], ReIGNN, uses GNNs and structural features to identify sFFs. The features include the number of gate types, the number of inputs, the number of outputs, and graph metrics, like the betweenness centrality and the harmonic centrality. The method additionally adds a post-processing that removes all FFs that are not part of any SCC.

3.5.3 General Feature Analysis

Table 3.1 lists all state-of-the-art methods of Section 3.5.2 to identify sFFs in a gate-level netlist, including the first two methods in this field [McE01; Shi+10], the similarity-based approaches RELIC [Mea+16], REWIND [Mea+18], RELIC-FUN [Gei+20], RELIC-Tarjan [Mea+19; MZJ19], fastRELIC [BBS19], the topological analysis [Fyr+18], and ReIGNN [CYN21]. It additionally contains a novel sFF identification enhancement [Bru+22a], which is presented in Chapter 4. The table shows what features each method uses for the identification and, thus, indicates methods with similar features or frequently applied features. The features of Table 3.1 are as follows:

Combinational FP: Some methods assume that an sFF has a combinational FP. They sort out any FFs which do not fulfill this property. Methods which use this feature: the first two methods in this field [McE01; Shi+10] and the topological analysis [Fyr+18].

FP of any class: Other methods assume that an sFF has an FP but do not further restrict its class, so the FP can be a generic, FSM, or combinational path. Methods which use this feature: fastRELIC [BBS19] and the novel sFF identification enhancement [Bru+22a].

Dependency on the same clock, reset, or enable signal: Some methods use FSM specific signals to derive an initial grouping of FFs. FFs which have the same clock, reset, or enable signals are sorted into the same groups. Another grouping criterion can be the type of FFs, e.g. separating synchronous from asynchronous FFs. Methods which use this feature: the second initial method [Shi+10] and the topological analysis [Fyr+18].

Influence on the design's control signals: Still, other methods assume that an sFF affects one or more control signal(s) of the design. The effect is measured by analyzing the connections to control gates or signals, like the select signal of multiplexers. Methods that use this feature are the same as the ones that use the dependency on the same clock, reset, or enable signal: the second initial method [Shi+10] and the topological analysis [Fyr+18].

Dissimilar gate-level structures of input cones: Several methods assume that sFFs have less similar input structures than nFFs. To measure the (dis)similarity, various methods compare the gate types or Boolean functions of the FFs' input cones using various techniques or circuit depths. Methods which use this feature: RELIC [Mea+16], REWIND [Mea+18], RELIC-FUN [Gei+20], RELIC-Tarjan [Mea+19; MZJ19], fastRELIC [BBS19], and the novel sFF identification enhancement [Bru+22a].

Influence or dependency: Many methods consider the influence or dependency of sFFs on each other within the gate-level netlist. Various methods assume various degrees of how loosely connected or strongly connected sFFs should be. This implies the newly derived definitions of weak and strong connectivity (Definitions 7 and 6), but also includes customized specifications, like that sFFs should form an SCC [e.g. MZJ19] or that strong sFFs with minimum generic paths should additionally have a strong connectivity with minimum combinational paths to a subset of all sFFs in the sFF set [Fyr+18]. Methods which use this feature: the second initial method [Shi+10], RELIC-Tarjan [Mea+19; MZJ19], the novel sFF identification enhancement [Bru+22a], the topological analysis [Fyr+18], and ReIGNN [CYN21].

Further structural features: Some methods use additional structural features besides the FP class, dissimilarity, or influence/dependency to identify sFFs. Such features include the

Table 3.1 Overview of sFF features which are exploited by state-of-the-art sFF identification methods.

Method	Combinational FP	Any FP	Grouping based on 'clock' or 'enable' or 'reset'	Effect on control signals	Dissimilarity	Influence/ dependency behavior or SCC	Other structural features
[McE01]	✓						
[Shi+10]	✓		✓	✓		✓	
[Mea+16; Mea+18; Gei+20]					✓		
[Mea+19; MZJ19]					✓	✓	✓
[BBS19]		✓			✓		
Chapter 4 ([Bru+22a])		✓			✓	✓	
[Fyr+18]	✓		✓	✓		✓	
[CYN21]						✓	✓

number of specific gate types in the sFF input cone, the number of inputs and outputs, the distance to external inputs and outputs, the distance to other FFs, or other graph metrics, like the betweenness centrality and the harmonic centrality. Methods which use such features: RELIC-Tarjan [Mea+19; MZJ19] and ReIGNN [CYN21].

Table 3.1 shows that the three most prominent features for sFF identification methods are the combinational FP, the dissimilarity, and the influence and dependency behavior. By analyzing and comparing state-of-the-art sFF identification methods, we identify two features that have a significant impact on the success of identifying sFFs and can be avoided during FSM design: combinational FP and dissimilarity (highlighted in Table 3.1). Section 5.3 shows how one can use these findings of prominent and exploitable features to design a novel FSM obfuscation technique. Section 5.3 also explains how the two features, combinational FP and dissimilarity, can be eliminated for an FSM design.

3.5.4 Classification Based on State Flip-Flop and State Flip-Flop Set Properties

Section 3.5.3 classifies sFF identification methods based on their applied features. In contrast, this section classifies sFF identification methods based on the predefined sFF set property categories in Section 3.4.1 which are derived by analyzing the degrees of freedom of FSMs in Section 3.2. An sFF identification method outputs an sFF set, which can be sorted into a category of Figure 3.5. Depending on what minimum path class and connectivity all sFFs of the sFF set fulfill, the sFF identification method will be classified. Instead of categorizing each sFF identification method of Section 3.5.2 individually, we summarize them into two main sFF identification strategies and then categorize these sFF identification strategies.

The two main strategies of sFF identification methods are similarity-based and SCC-based methods. Similarity-based methods use the similarity between FFs input cones as the main feature for identifying sFFs, like methods based on RELIC, see Section 3.5.2. SCC-based methods output sFFs which belong to the same SCC, but do not necessarily form an SCC, i.e. which fulfill SCC-property 1 but do not necessarily fulfill SCC-property 2, see Definition 13. This includes methods that either define sFFs based on SCCs, like RELIC-Tarjan, or split or sort out FFs based on SCCs, like the topological analysis. There exist other methods that do not entirely fulfill the conditions for a similarity-based or an SCC-based method. Nevertheless, by analyzing their features they can also be categorized in Figure 3.5.

Instead of categorizing individual methods, we can now categorize the two main strategies of sFF identification methods, similarity-based and SCC-based methods. The sFF results of similarity-based methods usually show no strict structural properties. The resulting sFF sets can contain sFFs which are neither weak nor strong sFFs and with arbitrary path class. Similarity-based methods are therefore classified into none of the defined categories, see Figure 3.8. Based on additional criteria of the methods, the properties of the resulting sFF set can be defined more precisely. The method fastRELIC, for example, removes all FFs which do not have an FP of minimum generic class. However, this restriction does not change the overall sFF properties to weak or strong connectivity.

The sFF results of SCC-based methods satisfy specific structural properties because the resulting sFFs belong to the same SCC but do not necessarily form an SCC. Such sFFs fulfill the SCC-property (1), i.e. the reachability requirements, but not the SCC-property (2), i.e. a maximal set. Consequently, sFF results of SCC-based methods are classified in the category with strong connectivity and generic path class, and fulfill the necessary condition of membership, see Figure 3.8. Again, based on additional criteria of the methods, the properties of the resulting sFF set can be defined even more precisely. Some SCC-based methods, like

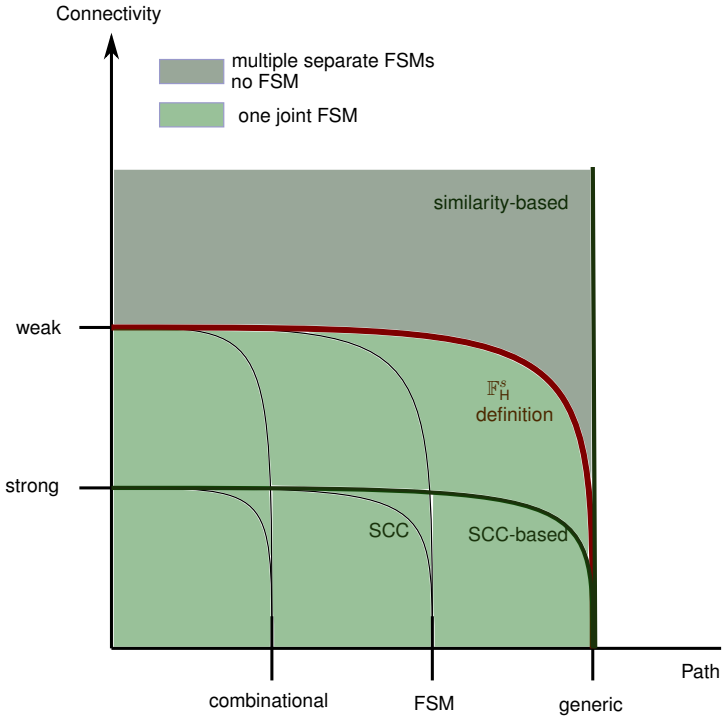


Figure 3.8 Classification of sFF identification methods based on sFF set categories. The labels of the categories mark the classification of the two main sFF identification strategies, “similarity-based” and “SCC-based”, as well as other specific sFF sets, including the SCC “SCC” (Definition 13), and the newly presented sFF set definition, “$\mathbb{F}_{\mathrm{H}}^{s}$ definition” (Definition 14).

the second part of the method in [Shi+10] or RELIC-Tarjan in [MZJ19], output sFFs which do not only belong to the same SCC but also form an SCC, i.e which do fulfill the SCC-properties (1) and (2). Consequently, they can also be classified in the category with strong connectivity and FSM path class, see Figure 3.8. The condition of membership depends on the FF set which is used to form the SCC. For [Shi+10], a subset of all FFs of the netlist is used, resulting in a necessary condition of membership, while for RELIC-Tarjan, all FFs of the netlist are used, resulting in a sufficient condition of membership. For other SCC-based methods, like the topological analysis in [Fyr+18], the additional criteria do not change the properties equally for all sFFs and thus do not change the categorization of the sFF set.

Methods that cannot be clearly assigned to one sFF identification strategy are ReIGNN [CYN21] and the novel sFF identification enhancement in Chapter 4 [Bru+22a]. ReIGNN sorts out FFs which do not belong to any SCC, but it does not divide the sFFs of the sFF set according to their membership in different SCCs. Consequently, the resulting sFF set can contain sFFs from multiple SCCs or FSMs and is therefore classified into none of the defined sFF set property categories. The novel sFF identification enhancement can be categorized into multiple categories, depending on what post-processing option is chosen. Section 4.1 gives a detailed categorization of this method.

3.6 Summary

The chapter deals with the features of FSMs. It starts with a general introduction of FSM basics and its terminology. Next, it further analyzes this definition and derives sFF and sFF set properties based on the identified degrees of freedom of the definition. Using the derived

properties, the chapter presents a systematic categorization of sFF sets, which can thus also be utilized to categorize sFF identification strategies. This categorization enables a statement to be made whether a set of sFFs is a valid sFF set for a state machine and whether a valid sFF set shows the required sFF and sFF set properties to be identifiable by a concrete sFF identification strategy. In addition, the categorization shows that SCC-based sFF identification strategies assume more restrictive sFF properties *connectivity* and *path* than similarity-based strategies. On top, the chapter presents a new sFF set definition for a human-readable state machine, which is sorted into the least restrictive category of the systematic categorization. This sFF set definition determines the minimum requirements of an FSM as they are assumed to exist for every designed FSM. Finally, the chapter presents a second option to classify sFF identification techniques: the features that an identification method assumes that an sFF or sFF set must have. The analysis shows that there exist features that are expected by several different identification methods.

4 Enhancements in State Flip-Flop Identification

This chapter presents the methodology of four newly developed post-processing algorithms and analyses their applicability. Also, the chapter shows a property-driven evaluation of sFF sets and sFF identification algorithms. This includes the categorizing of exemplary sFF sets, the results of the newly developed post-processing methods, and the analysis of the results of sFF identification techniques.

Except for minor adjustments, this chapter has already been pre-published in the author's works [Bru+22a] and the accelerated version of fastRELIC was implemented during the bachelor thesis of Maximilian Putz [Put19].

4.1 Post-Processing

The categorization of sFF identification strategies in Section 3.5.4 shows that there exist methods which identify sFF sets that do not fulfill the required sFF properties of $\mathbb{F}_{\mathsf{H}}^{s}$, see Definition 14 and Figure 3.8. Consequently, this section presents post-processing methods that shift these sFF sets into a category that possesses the desired sFF properties and thus predominantly improve sFF identification results. Unlike other post-processing-like techniques [e.g. CYN21; BBS19], the novel post-processing methods cope with the different sFF properties and are designed to be applied to the output of different identification methods. The section also analyzes the novel post-processing methods regarding their applicability on different types of sFF sets.

4.1.1 Methodology

In general, all sFF sets that do not already fulfill some desired properties can be post-processed. However, for sFF sets of similarity-based sFF identification strategies, post-processing might be required to at least fulfill the properties of $\mathbb{F}_{\mathsf{H}}^{s}$ because their outputs need not fulfill weak connectivity and generic path class, see Section 3.5.4. Without loss of generality, the following section focuses on post-processing applied to the results of similarity-based methods. Based on the sFF set property categories, four post-processing methods are derived: PP - weak,generic, PP - weak,FSM, PP - strong,generic, and PP - strong,FSM, see Figure 4.1. The first attribute denotes the connectivity, and the second attribute denotes the path class. Post-processing methods that target combinational paths are not considered, see Section 4.2.2 for further explanation.

Figure 4.1 shows the categorization of post-processing methods, similar to the categorization of sFF identification methods in Figure 3.8. Figure 4.1 enables an analysis of what post-processing methods satisfy the same minimum sFF properties as other specific sFF sets by investigating the labels that annotate the categories. The labels of the category weak connectivity and generic path class show that PP - weak,generic produces an sFF set whose sFFs satisfy the same minimum requirements as the sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$. The labels of the category strong connectivity and generic path class show that PP - strong,generic produces an sFF

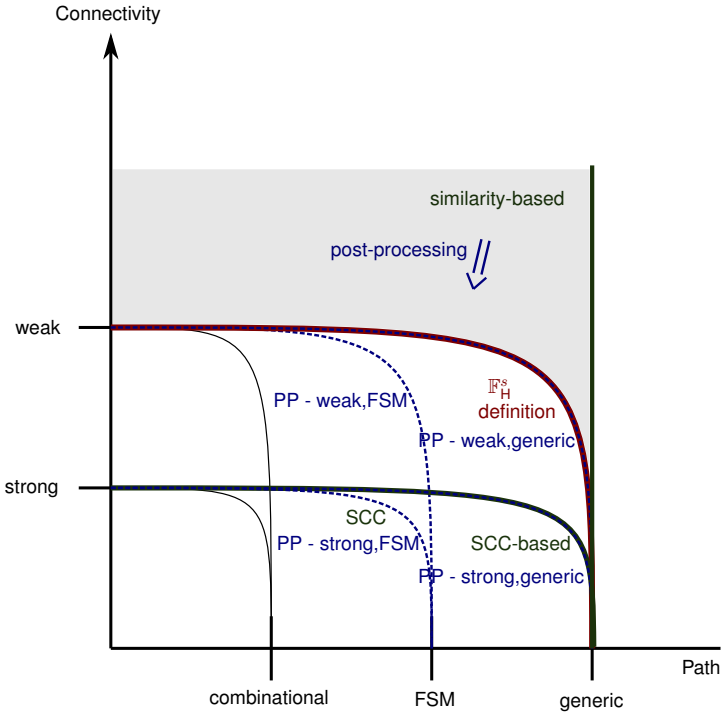


Figure 4.1 Overview of derived post-processing methods based on the sFF set categories in Figure 3.5. The categories weak and strong connectivity with generic and FSM path class are additionally labeled with the four post-processing techniques, PP - weak,generic, PP - weak,FSM, PP - strong,generic, and PP - strong,FSM (colored in blue) because their outputs minimum fulfill the corresponding connectivity and path conditions. The goal of post-processing methods is to shift the resulting sFF sets of sFF identification methods into a category with the desired sFF properties. The arrow in the gray-colored area outside the categories, labeled with “post-processing”, denotes this shift.

set whose sFFs satisfy the same minimum sFF properties as the sFF results of the SCC-based methods. The labels of the category strong connectivity and FSM path class show that PP - strong,FSM produces an sFF set whose sFFs satisfy the same minimum sFF properties as FFs forming an SCC. The labels of the category weak connectivity and FSM path class show that PP - weak,FSM does not correspond to any technique in Figure 3.8, which outputs sFFs with the same minimum sFF properties. The sFF sets generated by post-processing methods will always fulfill the necessary condition of membership and might even fulfill the sufficient condition of membership.

Post-processing - weak,generic

PP - weak,generic in Algorithm 1 outputs an sFF set which contains only weak or strong sFFs with minimum generic paths. Consequently, the minimum requirement for an sFF to be part of the resulting sFF set of PP - weak,generic is an FP and that some path exists between each pair of sFF in the set. Both paths have to be of minimum generic path class. As a first step of PP - weak,generic, the FPs of the identified sFFs are checked for a minimum generic class. The algorithm starts with the FP check because—in contrast to the cross-FF-influence check—the check for a minimum generic FP of an sFF is independent of the other sFFs in the sFF set. If an sFF does not have an FP of minimum generic class, i.e. if it does not have any FP at all, no adaption of the sFF set will be able to achieve the targeted property of minimum generic paths while keeping this sFF. Thus, the algorithm eliminates all identified sFFs which

Algorithm 1 Pseudocode of PP - weak,generic

Data: Set of FFs classified to be sFFs Set_{CFF}
Result: Updated set of classified sFFs after PP - weak,generic

```
1: function PP - WEAK,GENERIC(Set_CFF)
2:     for F_i ∈ Set_CFF do                                  ▷ Check FPs (generic)
3:         if no generic FP of F_i then
4:             Set_CFF.remove(F_i)
5:     while True do                                         ▷ Check cross-FF-influence
6:         counter[Set_CFF]=0
7:         for F_i ∈ Set_CFF do
8:             for F_j ∈ Set_CFF\{F_i} do
9:                 if no dependence (generic class) between F_i & F_j then
10:                    counter[F_i, F_j]++
11:        max_c=max(counter.values)
12:        if max_c==0 then
13:            break                                         ▷ Termination
14:        Set_CFF.remove(FF with counter equal to max_c)
15:    return Set_CFF
```

do not have an FP of minimum generic class, see lines 2-4 of Algorithm 1. This step will have no effect if similarity-based identification methods, such as fastRELIC, already consider such aspects. Next, the algorithm checks the cross-FF-influence of the remaining sFFs in the sFF set. Similar to the FP check, if there is no minimum generic path between two sFFs of the sFF set, i.e. if there is no path at all between two sFFs of the sFF set, no adaption of the sFF set will be able to achieve the targeted property of minimum generic paths while keeping both of these sFFs. While for the FP check, the required post-processing step is clear, this is not the case for the cross-FF-influence check. If there is no path between two sFFs, it has to be decided which of the two sFFs is eliminated from the originally classified sFF set. In order to make a preferably fair decision, first, the presence of all required influence paths is checked. The missing ones are documented by increasing a counter value for each of the relevant FFs, i.e. if an FF F_1 is reviewed and there is no path from F_1 to an FF F_2 and from F_2 to F_1, the counter values for F_1 and F_2 are increased by one, see lines 7-10 of Algorithm 1. Finally, the FF with the highest counter value is removed from the originally classified sFF set. If there are multiple FFs with the maximum counter value, one is removed arbitrarily. Next, the cross-FF-influence check (i.e. the counter calculation procedure) is repeated using the updated classified sFF set. The post-processing is terminated if all counter values are equal to zero, i.e. no sFFs remain without the required influence paths, or if the classified sFF set is empty. The termination criterion can also be weakened by ending the post-processing for counter values below a specific threshold greater than zero. However, this will allow FFs to stay in the classified sFF set, even if they do not fulfill the targeted sFF properties.

In the context of RE, the complexity and runtime of processes or algorithms often play a secondary role compared with the achieved RE success. Nevertheless, a short analysis is added for each post-processing algorithm. The complexity of Algorithm 1 strongly depends on the number of identified sFFs which are provided as input, because the dependency check in lines 9 and 10 is called $r = \mathcal{O}((\#\text{ identified sFFs})^3)$ times.

Algorithm 2 Pseudocode of PP - weak,FSM

Data: Set of FFs classified to be sFFs Set_{CFF}
Result: Updated set of classified sFFs after PP - weak,FSM

```
1:  function PP - WEAK,FSM(Set_CFF)
2:      for F_i ∈ Set_CFF do                                   ▷ Check FPs (generic)
3:          if no generic FP of F_i then
4:              Set_CFF.remove(F_i)
5:      for F_i ∈ Set_CFF.copy do                              ▷ Check FPs (FSM) + extend
6:          ▷ use updated Set_CFF for FSM path check ◁
7:          if no FSM FP of F_i then
8:              sP=shortest_feedback_path(F_i)
9:              Set_CFF.append(all FFs on sP ∉ Set_CFF)
10:     while True do                                          ▷ Check cross-FF-influence
11:         counter[Set_CFF]=0
12:         for F_i ∈ Set_CFF do
13:             for F_j ∈ Set_CFF\{F_i} do
14:                 if no dependence (FSM class) between F_i & F_j then
15:                     counter[F_i, F_j]++
16:         max_c=max(counter.values)
17:         if max_c==0 then
18:             break                                          ▷ Termination
19:         F_c=FF with counter equal to max_c
20:         Set_CFF.remove(F_c)
21:         affected_FF=[FFs of Set_CFF with F_c on previous FP]
22:         while len(affected_FF)!=0 do                       ▷ Check FPs of affected FFs
23:             for F_i ∈ affected_FF do
24:                 ▷ use updated Set_CFF for FSM path check ◁
25:                 if no FSM FP of F_i then
26:                     Set_CFF.remove(F_i)
27:                     affected_FF.append(FFs of Set_CFF with F_i on previous FP)
28:                     affected_FF.remove(F_i)
29:                     break
30:                 affected_FF.remove(F_i)
31:     return Set_CFF
```

Post-processing - weak,FSM

PP - weak,FSM in Algorithm 2 outputs an sFF set which contains only weak or strong sFFs with minimum FSM paths. The minimum requirement for an sFF to be part of the resulting sFF set of PP - weak,FSM is an FP and that some path exists between each pair of sFF in the set. Both paths have to be of minimum FSM path class. The post-processing procedure for a generic path class in Algorithm 1 is adapted accordingly, resulting in the pseudocode of Algorithm 2. The differences between the two pseudocode descriptions are marked in blue. The first FP check in Algorithm 1 is extended by a second one (see lines 5-9 in Algorithm 2), because—in contrast to the post-processing procedure for a generic path class—PP - weak,FSM requests not only the existence of FPs, but also that they are of minimum FSM path class. Still, for the same reason as for PP - weak,generic, sFFs without a minimum generic FP have to be eliminated from the sFF set. However, for an sFF with a generic, but no minimum FSM FP, an adaption of the sFF set will be able to achieve the targeted property of minimum FSM paths while keeping this sFF.

Thus, for all existing FPs, the algorithm checks if they are of minimum FSM path class and, if not, all previously nFFs which are located on the associated shortest FP are added to the classified sFF set. We choose the shortest FP to avoid adding a disproportionate number of FFs to the sFF set. If multiple shortest FPs exist, one is chosen arbitrarily. Those added FFs do not have to be checked for a suitable FP because they were added as components of a suitable FP. The second adaption for PP - weak,FSM is done during the counter calculation. The path, which must exist between each pair of sFFs in the set, must also satisfy a minimum FSM path class, see line 14 of Algorithm 2. Similar to the FP check, the cross-FF-influence check could remove sFFs which do not fulfill a cross-FF-influence with minimum generic path class and add additional FFs to the sFF set until all other sFFs fulfill a cross-FF-influence with minimum FSM path class. However, to avoid a disproportionate extension of the original sFF set, the algorithm does not add further FFs to the sFF set. Instead, it uses the FF removal strategy of PP - weak,generic for all sFFs which do not fulfill a cross-FF-influence with minimum FSM path class. Indeed, in contrast to the post-processing procedure for generic path class, removing an sFF with maximum counter value F_1 in the post-processing procedure for FSM path class has an impact on the FP check. If F_1 was part of an FP of FF F_2, this requires a re-assessment of the FP check of F_2. If the removal of F_1 results in F_2 having no FP of FSM path class anymore, F_2 will also have to be removed. Of course, this can again result in new FFs of the classified sFF set losing their FSM FP, see lines 21-30 of Algorithm 2. After performing the required removal of FFs so that all FFs in the classified sFF set have an FP with minimum FSM path class, the cross-FF-influence check procedure, starting from the counter value calculation, is repeated. The termination criterion of this post-processing procedure is the same as for the one in Algorithm 1, i.e. the highest counter value is equal to zero, or the classified sFF set is empty.

The complexity of Algorithm 2 is comparable with the one of Algorithm 1. In fact, several new steps are added, but the dependency check in lines 14 and 15 is still called $r = \mathcal{O}((\#\text{ identified sFFs})^3)$ times.

Post-processing - strong,generic/FSM

The post-processing methods PP - strong,generic and PP - strong,FSM output sFF sets which contain only strong sFFs with minimum generic paths for PP - strong,generic and minimum FSM paths for PP - strong,FSM. The minimum requirement for an sFF to be part of such a resulting sFF set is an FP and a path in each direction between each pair of sFF in the set. Both paths have to be of minimum generic path class for PP - strong,generic or of minimum FSM path class for PP - strong,FSM. The difference to PP - weak,generic and PP - weak,FSM is, therefore, the definition of the cross-FF-influence, what is also visible in the definitions of strong and weak sFFs. Post-processing methods which target weak connectivity increase the counter for FFs F_i and F_j, in line 10 of Algorithm 1 and line 15 of Algorithm 2 if no path exists from F_i to F_j and no path exists from F_j to F_i. Post-processing methods that target strong connectivity increase the counter for F_i and F_j if not two paths exist, one from F_i to F_j and one from F_j to F_i. In order to implement PP - strong,generic and PP - strong,FSM, these lines must be changed accordingly.

The complexities of PP - strong,generic and PP - strong,FSM are comparable with the complexities of PP - weak,generic and PP - weak,FSM. While the dependency check for PP - weak,generic and PP - weak,FSM might already be successful after reviewing only one path direction, the dependency check for PP - strong,generic and PP - strong,FSM can only be successful after reviewing both path directions. Nevertheless, the dependency check is still called $r = \mathcal{O}((\#\text{ identified sFFs})^3)$ times.

4.1.2 Applicability Analysis of Post-Processing Methods

The introduced post-processing procedures do not guarantee perfect results. They do not even guarantee improved results compared to the original sFF set. Still, the post-processing procedures guarantee that for a non-empty output set, the classified sFFs satisfy the targeted sFF properties of connectivity and path. However, it is expected that the results will improve overall in the end.

The success of the post-processing procedures mainly depends on the following factors:

1. The design characteristics.
2. The input sFF set on which the post-processing is applied.
3. The sFF set which is used to rate the success of the post-processing results.
4. Additional degrees of freedom during the implementation of the post-processing procedures.

The first factor is difficult to control and even more difficult to measure. The second factor is related to the success of the underlying sFF identification strategy and is strongly connected to the first factor. For instance, if the wrongly classified sFF set already satisfies the targeted sFF properties, post-processing methods will not change, and therefore also not improve, this sFF set. The sFF set which is used to rate the success of all results in this work is chosen to be $\mathbb{F}_{\mathrm{H}}^{s}$. The last factor arises due to degrees of freedom during the post-processing procedures. Some of them are listed below:

- If the counter calculation outputs multiple sFFs with the maximum counter value, one of them is chosen arbitrarily.
- If more than one shortest path exists, one of them is chosen arbitrarily in PP - weak,FSM and PP - strong,FSM.
- Depending on the order to check sFFs for a suitable FP, different previously nFFs might be added in PP - weak,FSM and PP - strong,FSM.

Using concrete designs in Section 4.2, Table 4.1, we can estimate the impact of the arbitrary choice of which sFF to remove on the overall success to predominantly be none or only small. In the early iterations of the cross-FF-influence check, a group of sFFs with maximum counter value is calculated. During later iterations of the algorithm, this group is completely or incompletely removed from the sFF set. If there is a remainder after the last iteration, it is more often one sFF and less often multiple sFFs. The arbitrary order of sFF removal decides which sFFs comprise the remainder. This remainder can only have an impact on the overall success of post-processing if the group, together with the corresponding affected FFs, was composed of both correct sFFs and correct nFFs.

The post-processing methods which target a minimum FSM path, PP - weak,FSM and PP - strong,FSM, not only remove sFFs from the sFF set, but can also add new elements, see line 9 of Algorithm 2. Consequently, the resulting sFF set can have a decreased number of wrongly identified sFFs and nFFs. This is an advantage over the post-processing methods, PP - weak,generic and PP - strong,generic, which can only decrease the number of wrongly identified sFFs, but not of wrongly identified nFFs. A second design characteristic is the decision which FF is removed next. The decision is based on the calculated counters and is designed to make it preferably fair because an FF, which does not fulfill the required properties for the majority of the remaining sFF set elements, is assumed to be the best candidate.

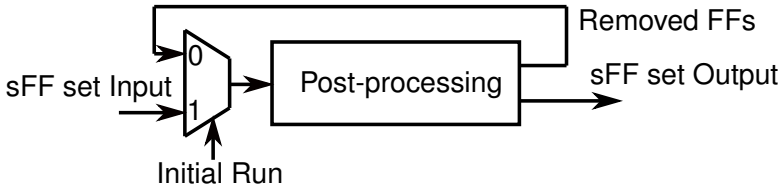


Figure 4.2 Applying post-processing recursively on eliminated FFs to extract multiple FSMs. The recursion stops if the sFF set output contains only one FF. This last sFF set will not be considered for the result.

However, in particular scenarios, this can also lead to less traceable decisions. For example, if PP - strong,FSM is applied on two groups that fulfill the required sFF properties individually but not together, always the larger group is chosen as sFF set output. However, this decision does not have to be the best solution.

For post-processing, it is assumed that the previously applied sFF identification method provides one or multiple sFF sets which are already near the correct single or multiple $\mathbb{F}_{\mathsf{H}}^{s}$. The post-processing methods are designed to improve the provided sFF sets separately. Nevertheless, post-processing methods have the potential to split FSMs because they sort out sFFs which do not fulfill the targeted connectivity and path properties. In sequential RE, separating a state machine into its original, multiple FSMs is a different research topic, and its success often depends on the level of connectivity between the single FSMs or sFF sets. Existing approaches make use of the assumption of weakly connected FSMs or common control signals to split the identified sFFs [e.g. Fyr+18; Shi+10] or the extracted FSM [e.g. Shi+10; MZJ16]. Consequently, if the post-processing method is applied recursively on the eliminated FFs, further FSMs which are sorted out in the first run might be extracted, see Figure 4.2.

4.2 Property-Driven Evaluation of State Flip-Flop Sets and State Flip-Flop Identification Methods

4.2.1 Testing Framework

A number of different benchmark circuits are synthesized using QFLOW [Ope19] and Yosys [Wol18], together with a reduced version of the $0.35\,\mu\mathrm{m}$ Oklahoma State University (OSU) library [Okl15]. The reduced library version contains AND gates, OR gates, Inverters, and synchronous and asynchronous FFs to ease the application of sFF identification methods. The chosen benchmarks include the well-known ITC99 benchmarks with an added reset (*) [CRS00], designs from OpenCores (†) [Ope], from secworks github (‡) [Str16; Str18b; Str18a], and from onchipuis github project mriscv (§) [Onc17a] and project mriscvcore (⋄) [Onc17b], see Tables 4.1 and 4.2. The characteristics of the benchmarks after the synthesis are also shown in the tables and cover the number of gates (without FFs) and the number of FFs. Table 4.1 additionally shows the number of sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$ (Definition 14), and the number of FFs of the FSM SCC, i.e. the SCC which contains most of the sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$. The last column of Table 4.1 indicates if $\mathbb{F}_{\mathsf{H}}^{s}$ is equal to the corresponding SCC, namely if the sFF sets of column five and six of Table 4.1 are identical. Instead, Table 4.2 states how many FSMs are part of the design and shows the number of sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$ (Definition 14) for each FSM.

The benchmarks in Table 4.1 are chosen in a way that they fulfill the previously made assumptions for $\mathbb{F}_{\mathsf{H}}^{s}$, i.e. all benchmarks contain exactly one FSM, and all sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$ are weak or

Table 4.1 Characteristics of benchmarks with a single FSM, sorted by number of FFs

	Design (single FSM)	# Gates	# FFs	# sFFs in $\mathbb{F}_{\mathsf{H}}^{s}$	# FSM SCC Elements	Identical Sets
α)	b02_reset*	30	4	3	3	✓
β)	b01_reset*	51	5	3	3	✓
γ)	b06_reset*	59	8	3	3	✓
δ)	fsm$\diamond$	136	15	7	7	
ϵ)	b10_reset*	235	17	4	11	
ζ)	b08_reset*	169	21	2	5	
η)	b09_reset*	172	28	2	19	
θ)	b11_reset*	630	31	4	25	
ι)	b07_reset*	458	49	3	11	
κ)	gpio$\S$	318	51	11	11	✓
λ)	b04_reset*	675	66	2	2	✓
μ)	mem$\diamond$	1130	75	7	7	✓
ν)	b12_reset*	1210	119	5	101	
ξ)	b14_reset*	5793	245	1	1	✓
o)	spi_axi$\S$	1631	555	9	9	✓
π)	siphash‡	11068	794	8	12	
ρ)	sha1_core‡	10385	850	3	10	
σ)	aes†	12720	901	16	20	
τ)	mem_controller†	12271	1079	66	537	
ϕ)	altor32_lite†	13885	1249	6	11	
χ)	fpSqrt†	22696	1364	3	11	
ψ)	cr_div†	46060	4172	4	2060	

Table 4.2 Characteristics of benchmarks with multiple FSMs

	Design (multiple FSMs)	# Gates	# FFs	# FSMs	# sFFs in $\mathbb{F}_{\mathsf{H}}^{s}$ per FSM
a)	b15_reset*	9100	449	2	(3,4)
b)	aes2‡	36568	2994	4	(3,4,4,4)

strong sFFs with minimum generic path class. Additionally, two benchmarks with multiple FSMs are added (Table 4.2) to show the applicability of the post-processing methods on designs with more than one FSM. For each individual FSM, the previously made assumptions for $\mathbb{F}_{\mathsf{H}}^{s}$ are again satisfied.

We evaluate the results using the well-known confusion matrix, see Table 4.3. Considering the sFF identification, “Positive” represents sFFs and “Negative” represents nFFs. Thus, “True Positive” and “True Negative” are the correctly identified sFFs and nFFs, while “False Positive” and “False Negative” are the wrongly identified sFFs and nFFs. In the following evaluation, correct sFFs are defined based on $\mathbb{F}_{\mathsf{H}}^{s}$.

4.2.2 Categorization of Exemplary State Flip-Flop Sets

This section analyzes and categorizes the structural properties of exemplary $\mathbb{F}_{\mathsf{H}}^{s}$ sFF sets. Table 4.4 considers the categories of weak connectivity and minimum generic, FSM and combinational path class. All designs fulfill the properties weak connectivity and generic

Table 4.3 Confusion matrix regarding the identification of sFFs

		Identified sFFs: Positive (P)	Identified sFFs: Negative (N)
Correct sFFs	Positive (CP)	True Positive (TP)	False Negative (FN)
	Negative (CN)	False Positive (FP)	True Negative (TN)

Table 4.4 Benchmarks in which the sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$ are weak or strong sFFs with minimum generic, FSM or combinational paths. Satisfied properties for a benchmark are marked with ✓in the associated table entry, not satisfied properties with blank space.

$\mathbb{F}_{\mathsf{H}}^{s}$ of benchmark	α)	β)	γ)	δ)	ϵ)	ζ)	η)	θ)	ι)	κ)	λ)	μ)	ν)	ξ)
Weak connectivity + path class														
minimum generic	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
minimum FSM	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
minimum combinational	✓	✓	✓		✓	✓	✓		✓		✓		✓	✓

$\mathbb{F}_{\mathsf{H}}^{s}$ of benchmark	o)	π)	ρ)	σ)	τ)	ϕ)	χ)	ψ)
Weak connectivity + path class								
minimum generic	✓	✓	✓	✓	✓	✓	✓	✓
minimum FSM	✓	✓	✓	✓	✓	✓	✓	✓
minimum combinational		✓			✓		✓	

path class (see the first row) because the designs are chosen accordingly, see Section 4.2.1. Additionally, all designs except *b11_reset* (θ) fulfill the properties weak connectivity and FSM path class (see the second row), and most of the designs fulfill the properties weak connectivity and combinational path class (see the third row).

The design *b11_reset* has one sFF with a generic, but no FSM or combinational FP. The reason for this is the optimization during the design synthesis. For further analysis of this case, we consider the bench file of *b11* [CRS00], which is provided by the designers and already contains the synthesized design *b11*. This bench file is also used as a starting point to generate the synthesized design *b11_reset* in Table 4.1. To do this, first, a reset signal is added using a script provided by the designers for this purpose, and second, it is synthesized by the chosen synthesis tools of this work, QFLOW and Yosys. These two steps do not change the sFF properties in Table 4.4 of the synthesized FSM of *b11*. Thus, the following description is based on the unprocessed bench file of *b11*. The FSM in this design consists of nine binary-encoded states and has thus four state bits or sFFs. The sFF with a generic but no FSM or combinational FP is the fourth state bit. Due to the binary encoding, it is evaluated to zero for all but the state *1000*. The state *1000* has a single transition: *1000* $\rightarrow$ *0001*. There exists another state in the FSM which also has only one transition and whose single transition goes to *0001*: the state *0000*. The encoding of state *0000* differs from the encoding of state *1000* only in its fourth bit. Consequently, the assignment of the next value of the fourth state bit does not directly depend on its current value but only on other signals, like the values of the remaining sFFs or nFFs. Also, the fourth state bit does not have to influence itself or other sFFs directly. The synthesis detects these relations and optimizes the netlist accordingly. This results in the fourth sFF having no FSM or combinational FPs, but only a generic FP across nFFs, or FSM outputs and FSM inputs.

Table 4.5 considers the categories of strong connectivity and minimum generic, FSM and combinational path class. It shows that for strong connectivity, the $\mathbb{F}_{\mathsf{H}}^{s}$ of two designs do not fulfill

Table 4.5 Benchmarks in which the sFFs of $\mathbb{F}_{\mathsf{H}}^{s}$ are strong sFFs with a minimum generic, FSM or combinational path. Satisfied properties for a benchmark are marked with ✓in the associated table entry, not satisfied properties with blank space.

$\mathbb{F}_{\mathsf{H}}^{s}$ of benchmark	α)	β)	γ)	δ)	ϵ)	ζ)	η)	θ)	ι)	κ)	λ)	μ)	ν)	ξ)
Strong connectivity + path class														
minimum generic	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
minimum FSM	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓
minimum combinational	✓	✓	✓		✓	✓	✓		✓		✓		✓	✓

$\mathbb{F}_{\mathsf{H}}^{s}$ of benchmark	o)	π)	ρ)	σ)	τ)	ϕ)	χ)	ψ)
Strong connectivity + path class								
minimum generic	✓	✓	✓	✓	✓		✓	✓
minimum FSM	✓	✓	✓	✓	✓		✓	✓
minimum combinational					✓			

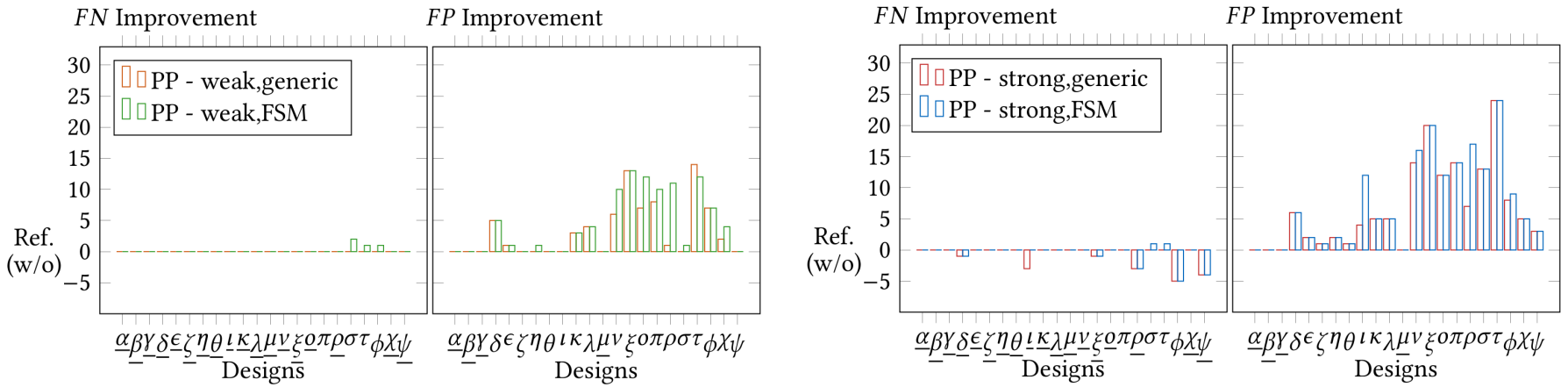
the sFF property of minimum generic path class (see the first row). The number of benchmarks of which the $\mathbb{F}_{\mathsf{H}}^{s}$ do not fulfill a strong connectivity and a minimum FSM or combinational path class increases respectively. Considering the properties of the second row (strong connectivity and minimum FSM path class), there is another design, *b11_reset* (θ), that no longer fulfills these properties. The argumentation is the same as for the results in Table 4.4. About half of the designs fulfill the properties strong connectivity and combinational path class (see the third row).

Both Tables 4.4 and 4.5 show that compared to other path classes, significantly fewer designs have a $\mathbb{F}_{\mathsf{H}}^{s}$ which fulfills the combinational path class. Consequently, post-processing methods that target combinational paths were not considered in Section 4.1. Summarizing, the Tables 4.4 and 4.5 show the associated categories of $\mathbb{F}_{\mathsf{H}}^{s}$ for each benchmark in Table 4.1.

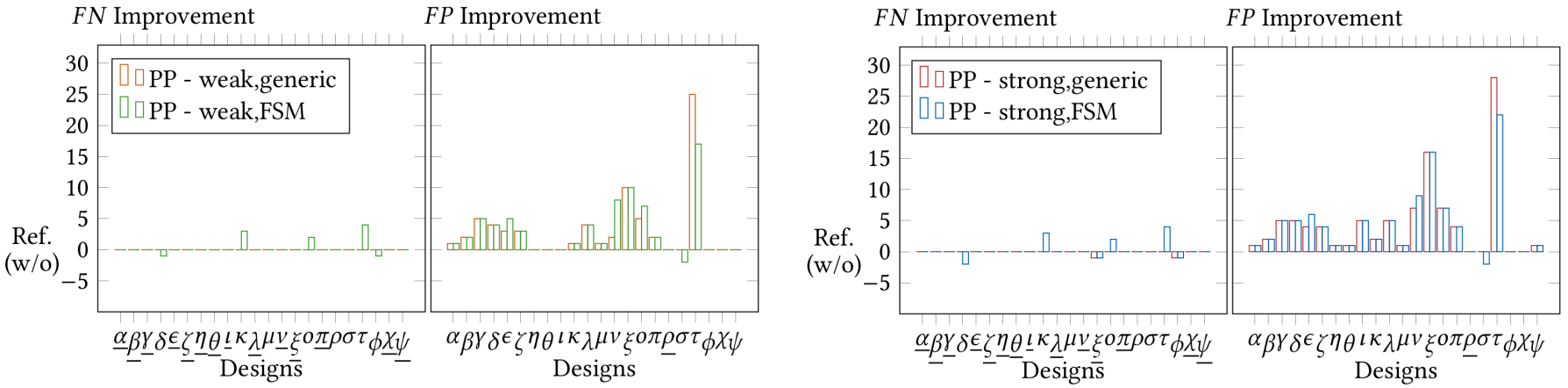
4.2.3 Evaluation of Post-Processing Methods

To evaluate the post-processing methods, similarity-based methods are applied first. We choose two similarity-based methods: an accelerated version of fastRELIC [BBS19], and *relic_o* of the toolset NetA [MZJ19]. For both methods, several different thresholds and parameters must be chosen [BBS19; MZJ19; Mea+16]. The parameter choice for the most successful results strongly relates to the underlying design. For simplicity and comparison reasons, we choose identical parameters for all designs, similar for fastRELIC and *relic_o*, and pick them heuristically to give a good result. Additionally, the asynchronous FFs are replaced by corresponding synchronous FF structures to calculate the similarity scores. For fastRELIC, we choose the parameters to be 0.7 for threshold $T1$, 0.8 for threshold $T2$, 2 for threshold ($T3$+1) and 5 for depth d. Our fastRELIC implementation defines the FF with the lowest index number in the design netlist as the starting point and determines potential sFFs separately for each clock region and combines them afterward to reduce the computational complexity. The second method, *relic_o*, uses a threshold equal to 0.7 and a depth of 5. The results are interpreted similarly to the groups of fastRELIC, based on the threshold $(T3+1) = 2$.

To evaluate the results of the different identification strategies and of the different post-processing techniques, $\mathbb{F}_{\mathsf{H}}^{s}$ is used as true condition. We compare the success by considering the FN and FP.



(a) fastRELIC



(b) relic_o

Figure 4.3 FN and FP improvement over all benchmarks based on $\mathbb{F}^s_H$, using the identification method fastRELIC and relic_o with all four post-processing methods. The results are compared against the reference 0, which is defined as the identification strategy without post-processing. Designs that achieve already zero FN or FP by the reference identification algorithm are underlined.

Figure 4.3 shows the FN and FP improvement over all designs in Table 4.1, for all post-processing methods, for fastRELIC and *relic_o*. If Figure 4.3 depicts positive numbers for the FN or FP improvement, the post-processing method achieved better FN or FP results compared with the initially applied identification method (marked as "Ref.(w/o)"), i.e. the number of FN or FP were decreased. If Figure 4.3 depicts negative numbers for the FN or FP improvement, the post-processing method achieved worse FN or FP results compared with the initially applied identification method (marked as "Ref.(w/o)"), i.e. the number of FN or FP were increased. If Figure 4.3 depicts zero FN or FP improvement, the post-processing method achieved the same FN or FP results compared with the initially applied identification method (marked as "Ref.(w/o)"), i.e. the number of FN or FP did not change. Note that the results for fastRELIC and *relic_o*, must not be compared. The parameters of the algorithms were chosen heuristically and might not truly reflect the capabilities of either method. The focus of Figure 4.3 is on the improvement brought about by the post-processing. Thus, for the following analysis, the results of fastRELIC and *relic_o* are not considered separately but together. Designs that already achieved a perfect FN or FP with fastRELIC or *relic_o* are underlined to differentiate between these designs and the designs for which no improvement was obtained.

The following analysis first concentrates on the FN results. Figure 4.3 verifies the previously made assertions that the FN results can only be improved when using post-processing to achieve minimum FSM paths, namely PP - weak,FSM and PP - strong,FSM. PP - weak,FSM increases FN for two designs (negative numbers for FN improvement in Figure 4.3b for *fsm* (δ) and *altor32_lite* (ϕ)) but also decreases it for six designs (positive numbers for FN improvement in Figure 4.3a for *aes* (σ), *mem_controller* (τ), and *altor32_lite* (ϕ) and in Figure 4.3b for *gpio* (κ), *spi_axi* (o), and *mem_controller* (τ)). Instead, PP - strong,FSM predominantly increases FN (eight negative numbers versus five positive numbers for FN improvement in Figure 4.3). Using minimum generic paths always leads to similar or worse FN results, regardless of whether a weak or strong connectivity is targeted. PP - weak,generic does not change the FN results for any design, while PP - strong,generic increases FN for eight designs. Zero FN is usually preferable to zero FP, what promotes PP - weak,generic because this is the only post-processing method that does not increase FN for any of the example designs, i.e. which does not achieve worse FN results for any of the example designs.

When concentrating on the FP results, Figure 4.3 shows that FP is improved independently of the path class. Post-processing for both generic and FSM path classes lead to similar FP improvements (compare the green and the orange colored bars for post-processing methods which target weak connectivity and the red and blue colored bars for post-processing methods which target strong connectivity). Instead, the sFF property connectivity influences the FP improvement, with strong connectivity leading to better results (compare the left bar diagram showing the FP improvement and the right bar diagram showing the FP improvement in Figures 4.3a and 4.3b).

This trade-off between post-processing which targets weak and post-processing which targets strong connectivity is no longer visible if we use a different strategy to compare the four post-processing methods. For each design, we determine which post-processing method(s) achieve the best result. The best result is defined to have the lowest value of FN, or for equal FN values, the lowest FP value. For each post-processing method, we count how often it is selected. As a result, post-processing which targets strong connectivity is selected more often than post-processing which targets weak connectivity. PP - weak,generic is the least likely to be chosen, while PP - strong,FSM is the most common.

In summary, post-processing methods that target weak connectivity showed less FN degradations but also less FP improvements than post-processing methods that target strong connectivity. In contrast to post-processing methods that target the generic path class, post-processing methods that target FSM path class allow FN improvements. In general, one can start by applying PP - weak,generic because this post-processing method showed no FN and no FP degradations for all tested designs, see Figure 4.3. RE is an iterative process, so if the post-processing results seem not reasonable, one can choose a more restrictive post-processing method or use previous knowledge of the targeted $\mathbb{F}_{\mathsf{H}}^{s}$ if available. If the category of the targeted $\mathbb{F}_{\mathsf{H}}^{s}$ is known, see Table 4.4 and 4.5 in Section 4.2.2, one can choose the corresponding post-processing method to improve the results. The design *b02_reset* (α), for example, fulfills the conditions of all categories. Consequently, the most restrictive post-processing method, PP - strong,FSM, could also be applied. Instead, *fsm* (δ) does only satisfy the definitions of weak connectivity and generic or FSM path class, denoting the suitable usage of PP - weak,FSM.

Finally, we evaluate the results for the two example designs with multiple FSMs, *b15_reset* (a) and *aes2* (b), see Table 4.2. The identification method fastRELIC outputs an sFF set which contains all sFFs of both FSMs for *b15_reset* and all sFFs of all four FSMs for *aes2*. For *b15_reset*, post-processing reduces the number of FP, but does not sort out any of the sFFs of the two $\mathbb{F}_{\mathsf{H}}^{s}$ sFF sets, because the sets are connected very strongly. For *aes2*, PP - weak,generic and PP - weak,FSM sort out two $\mathbb{F}_{\mathsf{H}}^{s}$ sFF sets, while PP - strong,generic and PP - strong,FSM sort out three $\mathbb{F}_{\mathsf{H}}^{s}$ sFF sets, because the sets are connected very weakly. Additionally, also the number of FP is reduced. If the post-processing method is applied recursively on the eliminated FFs of *aes2* (see Section 4.1.2), PP - weak,generic or PP - weak,FSM result in four extracted sFF sets: one set with two $\mathbb{F}_{\mathsf{H}}^{s}$, two sets with one $\mathbb{F}_{\mathsf{H}}^{s}$, and one set with no $\mathbb{F}_{\mathsf{H}}^{s}$. PP - strong,generic or PP - strong,FSM result in six extracted sFF sets: four sets with one $\mathbb{F}_{\mathsf{H}}^{s}$, and two sets with no $\mathbb{F}_{\mathsf{H}}^{s}$.

4.2.4 Evaluation of State Flip-Flop Identification Strategies

This section evaluates and compares the two sFF identification strategies, the SCC-based and the similarity-based sFF identification strategy, based on $\mathbb{F}_{\mathsf{H}}^{s}$, considering the newly derived systematic categorization in Section 3.4.1. For the similarity-based method, we choose the accelerated version of fastRELIC [BBS19], and for the SCC-based method, we choose RELIC-Tarjan [MZJ19; Mea+19]. As a small adjustment for RELIC-Tarjan, the signal with the highest Z-score value is chosen, which leads to an SCC with a minimum of two, instead of one, FF signals as elements. This results in an sFF set which forms an SCC, and consequently satisfies strong connectivity and minimum FSM path class, see Section 3.5.4. Thus, we choose PP - strong,FSM as the post-processing method for comparison reasons. The results for the remaining post-processing methods are shown in Appendix A.1.

As before, several different thresholds and parameters are chosen for the two strategies [BBS19; MZJ19; Mea+16]. For fastRELIC, we use the parameters as before. For RELIC-Tarjan, more precisely for the NetA tool *relic*, no number of components and no features were provided, i.e. the default values for these parameters were kept. Only the shift parameter *shft* is chosen based on the design specific nFF to sFF ratio, see Table 4.6.

Table 4.7 shows the results for the similarity-based method, fastRELIC, the SCC-based method, RELIC-Tarjan, as well as PP - strong,FSM which is applied to the sFF set output of fastRELIC. Additionally, a column is added that takes previous knowledge about the sFF set into account. Assuming we know what connectivity and path class the targeted $\mathbb{F}_{\mathsf{H}}^{s}$ of a specific

Table 4.6 Value for the parameter *shft* of the NetA tool *relic* based on nFF to sFF ratio range.

ratio range	<10	<50	<100	<500	<1000	≥ 1000
shft	0	1	5	10	50	100

benchmark fulfills, i.e. to which sFF set categories the targeted $\mathbb{F}_{\mathsf{H}}^{s}$ belongs to, we can choose the best suitable post-processing technique. As sFF set property categories are not mutually exclusive, $\mathbb{F}_{\mathsf{H}}^{s}$ can belong to more than one category, see Figure 3.5 and Tables 4.4 and 4.5. We always choose the most restrictive category, i.e. the category that includes the least other categories, to determine which post-processing method to use, see Figure 4.1. For all but the designs *fsm* (δ), *b11_reset* (θ) and *altor32_lite* (ϕ), this is PP - strong,FSM, see Tables 4.4 and 4.5. For *b11_reset*, PP - strong,generic and for *fsm* and *altor32_lite*, PP - weak,FSM is chosen, respectively. The design *b14_reset* (ξ) is excluded from further analysis because its FSM only consists of one sFF, which is not covered by the adjusted RELIC-Tarjan. To compare the two sFF identification strategies, the success of fastRELIC versus RELIC-Tarjan, and the success of the chosen post-processing method versus RELIC-Tarjan is investigated. A more successful result is defined as a result with a smaller FN value, or for equal FN values, a result with a smaller FP value. fastRELIC provides a better result than RELIC-Tarjan for six designs, see green colored cells in Table 4.7, and a worse result for eleven designs, see red colored cells in Table 4.7. PP - strong,FSM is better for five and worse for five designs; this is a balanced result for this set of benchmarks. If previous knowledge about the targeted $\mathbb{F}_{\mathsf{H}}^{s}$ is assumed, a corresponding post-processing method can be chosen for each. This leads to better results for seven designs and worse for four, allowing for a better overall result for this set of benchmarks.

To further understand in which cases a similarity-score-based approach with post-processing is unable to perform better than the SCC-based approach, these four designs are considered in more detail. For the first design, *siphash* (π), fastRELIC cannot classify three of the eight sFFs with the tested parameter set, and while PP - strong,generic and PP - strong,FSM can reduce the number of FP, the missing sFFs cannot be added. For the designs *sha1_core* (ρ) and *cr_div* (ψ), the scenario described in Section 4.1.2 occurs: The correct sFFs belong to an SCC and to the classified sFF set by fastRELIC, see zero FN for the fastRELIC column in Table 4.7. However, there exists an additional larger SCC in the classified set, making it impossible to find only the smaller, correct sFF set. As a result, post-processing leads to a larger, wrong sFF set, see the FN values of the post-processing methods. Finally, fastRELIC is unable to identify 51 of the 66 correct sFFs for the *mem_controller* (τ), while RELIC-Tarjan is able to identify all sFFs correctly, at the cost of an extremely high number of FP. PP - strong,FSM can reduce the number of FP for fastRELIC, but cannot classify the additional missing sFFs. Neither method is ideal for this type of design, as neither will lead to an extraction of an sFF set for a human-readable FSM.

Influence of State Flip-Flop Property Connectivity

The sFF property connectivity has a direct influence on which sFFs can be identified by specific sFF identification strategies or post-processing methods, see Figure 4.1. The methods that output sFF sets that are sorted in the most restrictive categories, i.e. of strong connectivity and minimum FSM or combinational path, are often SCC-based methods, such as RELIC-Tarjan, and PP - strong,FSM. Both will only identify sFFs if they are strong sFFs with a minimum FSM path. An SCC-based method or PP - strong,FSM/generic will not identify a weak sFF, namely an sFF which is either only influenced by or only influences all the other sFFs. Such a case

Table 4.7 FN and FP based on $\mathbb{F}^s_{\mathsf{H}}$ for an SCC-based identification method, RELIC-Tarjan (disregarding SCCs with only one FF), and a similarity-based identification method, fastRELIC, with and without PP - strong,FSM and the post-processing method according to the category of the targeted $\mathbb{F}^s_{\mathsf{H}}$ of a specific design. Green/red colored cells mark a similarity-based method result which is better/worse than the corresponding SCC-based method result.

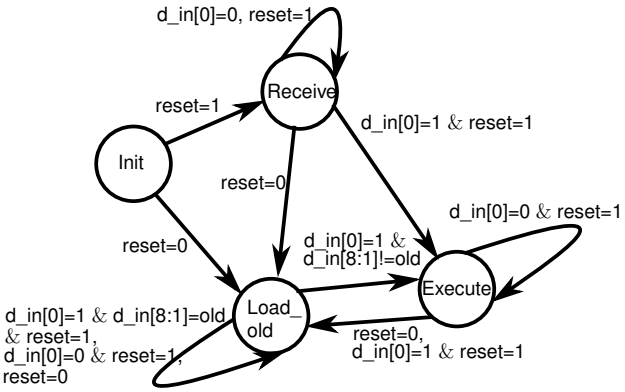
Design	fastRELIC (similarity-based)		RELIC-Tarjan (SCC-based)		PP - strong,FSM		Category-chosen post-processing	
	FN	FP	FN	FP	FN	FP	FN	FP
α)	0	0	0	0	0	0	0	0
β)	0	0	0	0	0	0	0	0
γ)	0	0	0	0	0	0	0	0
δ)	0	7	1	1	1	1	0	2
ϵ)	0	9	0	7	0	7	0	7
ζ)	0	4	0	3	0	3	0	3
η)	0	5	0	17	0	3	0	3
θ)	0	16	0	21	0	15	0	15
ι)	0	16	0	8	0	4	0	4
κ)	0	5	0	0	0	0	0	0
λ)	0	5	0	0	0	0	0	0
μ)	0	0	0	0	0	0	0	0
ν)	0	37	0	96	0	21	0	21
o)	0	12	0	0	0	0	0	0
π)	3	18	0	4	3	4	3	4
ρ)	0	43	0	7	3	26	3	26
σ)	9	17	16	720	8	4	8	4
τ)	51	111	0	471	50	87	50	87
ϕ)	1	20	1	6	6	11	0	13
χ)	0	13	0	8	0	8	0	8
ψ)	0	24	0	2056	4	21	4	21

appears in two benchmarks in Table 4.1: *fsm* (δ) and *altor32_lite* (ϕ), see Tables 4.4 and 4.5 in Section 4.2.2.

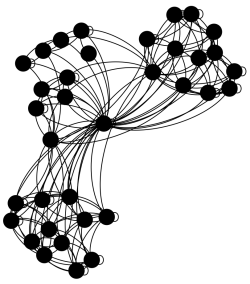
The state machine of the design *fsm* has a trap state with a specific trap state bit. The trap state can only be left if the reset signal is set to high. Consequently, the trap state bit does not influence any other state bits and is, therefore, a weak sFF. The state machine of the design *altor32_lite* has a reset state with a specific reset state bit. The reset state is only reached if the reset signal is set to high and only held if the enable signal is set to high. Consequently, the reset state bit is not influenced by any other state bits and is, therefore, a weak sFF. As a result, for both designs, RELIC-Tarjan and PP - strong,FSM do not identify the weak sFF, resulting in non-zero FN, see Table 4.7. Instead, fastRELIC or the post-processing method which is chosen according to the category of the targeted $\mathbb{F}^s_{\mathsf{H}}$ identify the weak sFF, resulting in zero FN.

Influence of State Flip-Flop Set Property

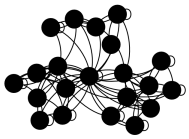
Considering the sFF set property condition of membership, the sizes of $\mathbb{F}^s_{\mathsf{N}}$ and $\mathbb{F}^s_{\mathsf{S}}$ (Definitions 11 and 12) can vary significantly. Table 4.1, columns five and six, exemplary can show this size difference because $\mathbb{F}^s_{\mathsf{H}}$ fulfills the necessary condition of membership with weak connectivity and an SCC fulfills the sufficient condition of membership with strong connectivity. In particular,



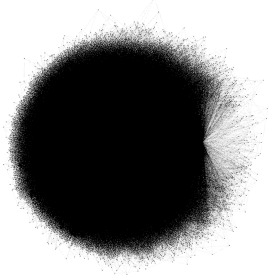
(a) Original FSM



(b) Extracted FSM using the identified sFFs of fastRELIC



(c) Extracted FSM using the identified sFFs of PP - strong,FSM



(d) Extracted FSM using the identified sFFs of RELIC-Tarjan

Figure 4.4 Original FSM and extracted FSMs of the design *b09_reset* in Algorithm 4, Appendix A.2, using the identified sFFs of fastRELIC, PP - strong,FSM, and RELIC-Tarjan, resulting in 35, 21 and 262401 FSM states. For better illustration, the state and transition labels have been omitted for subfigures (b), (c), and (d). The figures of the extracted FSMs are generated with the NetA tool *refsm* and option *norst* [Mea+19; MZJ19], as well as the network analysis and visualization software Gephi [BHJ09].

the effect will be nullified if the sets are identical (see column seven) and noticeable if the number of FFs in the corresponding SCC is significantly larger than the number of sFFs in $\mathbb{F}^s_{\mathsf{H}}$. Some examples of benchmarks with such large discrepancies are *b09_reset* (η), *b12_reset* (ν), or *cr_div* (ψ). These designs are characterized by intermediate signals, often comparatively larger than FSM state signals, which influence FSM transitions and are influenced by FSM states. Consequently, these signals are part of the same SCC as most or all of the sFFs in $\mathbb{F}^s_{\mathsf{H}}$.

The following analyzes the benchmark *b09_reset* (η) in more detail, see the excerpt of the pseudocode of *b09_reset* in Algorithm 4, Appendix A.2 and the corresponding FSM in Figure 4.4a. Its $\mathbb{F}^s_{\mathsf{H}}$ contains the FFs which represent the register *stato* after synthesis, resulting in two sFFs. The corresponding SCC contains both sFFs of $\mathbb{F}^s_{\mathsf{H}}$ and an additional 17 FFs. The concrete results of the similarity-based method fastRELIC and the SCC-based method RELIC-Tarjan reflect this discrepancy because $\mathbb{F}^s_{\mathsf{H}}$ is a subset of both of the resulting sFF sets and because of the sFF set property of fastRELIC and RELIC-Tarjan. The similarity-based method fastRELIC is classified into none of the sFF set categories and therefore usually cannot satisfy the sufficient condition of membership, whereas RELIC-Tarjan satisfies the sufficient condition of membership with strong connectivity, see Section 3.5.4. Both, fastRELIC and RELIC-Tarjan, result in sFF sets with no FN, but the sFF set of RELIC-Tarjan evaluates to 17 FP while the sFF set of fastRELIC evaluates only to five FP. The 17 FP of RELIC-Tarjan arise from the 8-bit register *old* and the 9-bit register *d_in*, see Algorithm 4 in Appendix

A.2. Together with the register *stato*, these three registers form an SCC. The five FP of fastRELIC consist of three FFs which are and two FFs which are not part of the corresponding SCC. PP - strong,generic and PP - strong,FSM remove the two FFs which are not part of the corresponding SCC, resulting in an sFF set with only three FP, see Table 4.7. The number of states in the corresponding state sets of the sFF sets of RELIC-Tarjan, fastRELIC and PP - strong,FSM are determined by applying the NetA tool *refsm* with the option *norst* [MZJ19; Mea+19] on these sFF set outputs. In general, the removal of FP allows to drastically reduce the number of FSM states. For the example *b09_reset*, the extracted FSMs have a strongly different number of states. RELIC-Tarjan produces an FSM with 262401 states. This can be reduced to 35 states by choosing fastRELIC instead (removing 12 FP) and to 21 states with additional PP - strong,FSM (removing another 2 FP). All three extracted FSMs contain the correct functionality but differ significantly in their human-readability, see Figure 4.4. Thus, for some designs, the sFF set property condition of membership that is followed by sFF identification algorithms can crucially impact the RE results.

4.3 Summary

This chapter shows the development of four post-processing methods. The post-processing methods cope with the different sFF properties and can combine the advantages of the less restrictive similarity-based sFF identification methods with the advantages of SCC-based methods which already make use of sFF properties (however quite restrictive ones). They adjust the sFF set output of similarity-based sFF identification methods such that the sFFs of the resulting sFF sets satisfy specific sFF properties. As a consequence, their results experience similar pros and cons as SCC-based sFF identification methods. However, they can benefit from a necessary, instead of from a sufficient condition of membership, and from the opportunity that a reverse engineer can select one out of four post-processing methods, i.e. targeted sFF properties, or can apply several of them one after the other. What post-processing method leads to the most successful result is, among others, strongly dependent on the underlying FSM, design, and input, and not generally predictable. However, at least for the chosen example designs, post-processing the similarity-based methods fastRELIC and relic_o overall predominantly leads to improved results. In addition, the results show that the more restrictive SCC-based sFF identification strategies cannot successfully identify all sFFs of the correct sFF set if the correct sFF set does not fulfill the assumed restrictive properties. Additionally, if a method assumes a sufficient condition of membership, the resulting sFF set can lead to FSMs with an enormous number of states, thus to non-human-readable FSMs. Overall, we conclude that combining different techniques, like a similarity-based identification method and a connectivity and path class controlled post-processing, can be a promising approach to improve RE results.

5 Novel State Machine Obfuscation Techniques

This chapter first gives an overview of state machine obfuscation strategies by classifying them into the three major threat models. Next, two novel FSM obfuscation strategies are presented: Timing Camouflage enabled FSM obfuscation, which is based on a recent camouflaging technique, Timing Camouflage; and a two-stage FSM obfuscation using attractive hardware honeypots and unattractive original FSMs to entangle state-of-the-art RE methods.

Parts of this chapter have already been pre-published in the author's works [Bru+22b] and [Bru+23]. Section 5.2 is based on [Bru+22b], and its implementation of the redesign approach, Redesign_Func, was supported by the bachelor thesis of Geralda Syku [Syk21]. Section 5.3 is based on [Bru+23], and its concept was supported by the master thesis of Hye Hyun Lee [Lee22], while the topological analysis tool was implemented by the working student Maximilian Putz. One of the approaches to generate an unattractive FSM, the similarity approach, in Section 5.3.2 is based on results of the author's master thesis [Bru18], but was majorly revised and extended during the author's Ph.D. studies. The chapter is extended by a novel part, Section 5.1, to introduce FSM obfuscation strategies and their threat models.

5.1 Overview of State Machine Obfuscation Strategies

This section concentrates on techniques that were explicitly developed to protect FSMs or sFFs. It does not elaborate on the fact that remediation strategies that target other parts of the netlist, e.g. scan path encryption [e.g. KCK20] or locking mechanisms for combinational netlists [e.g. RKM08a], might also affect the security of FSMs or sFFs. Instead, Section 6.2.1 gives an overview of state-of-the-art locking techniques in more detail.

When developing FSM obfuscation strategies, one first has to define against which threat model the novel technique should be effective. For simplification reasons, we only consider two possible attackers: the end user and the foundry. For other attackers, the requirements are very similar to the ones of the considered attackers, or they are assumed to be less likely to appear. For example, the testing or packaging company has similar requirements to the foundry. The designer, IP provider, or synthesis tool provider are extremely powerful attackers but less likely. Regarding FSM obfuscation methods, the work in [JS21a] differentiates two threat models, while we add a third one:

1. The unauthorized usage of the FSM by the end user and/or foundry.
2. The extraction or RE of the functionality of the FSM by the end user and/or foundry.
3. The malicious manipulation of the FSM by the end user and/or foundry, e.g. by modifying the circuit to insert an HT or by injecting faults to change the control flow and thus attack the system.

Unauthorized Usage (First Threat Model): For the first threat model, the unauthorized usage of the FSM, many locking-based methods exist, i.e. methods which require a correctly applied secret key or input pattern to enable the correct functionality, e.g. [Des+13; KJC19;

LNO21; LO22; Ros+20; RB22; Hu+20; Rah+23; KMV20; JS18; JS21a; JS21b; Pat+18; Yas+17c; LZ13; CB09; CNB13; DY18; MZJ17; KTV19; KV20]. Without the correct key bit or input sequence, the functionality or output behavior of the FSM is modified, e.g. [Des+13; KJC19; LNO21; LO22; Ros+20; RB22; Hu+20; Rah+23; KMV20; JS18; JS21a; JS21b; Pat+18], slowed down, e.g. [Yas+17c; LZ13], intentionally blocked, e.g. [CB09; CNB13; DY18; MZJ17; JS18; JS21a; JS21b], or a combination of these. Some methods use FSMs to corrupt an arbitrary circuit signal if the wrong key is provided and—similar to the FSM obfuscation technique in [Ros+20]—a rarely occurring state is reached [KTV19; KV20]. The appearance of such rare states is facilitated by adding counters to the original FSMs. Depending on how the key management is organized, methods for the first threat model mostly target the prevention of both attackers, the end user and the foundry.

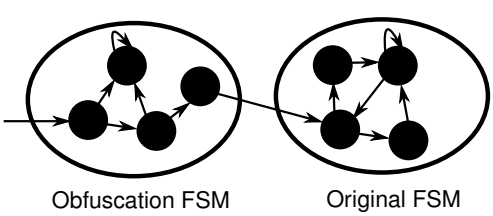
The threat model of unauthorized usage of the FSM is related to a different—not necessarily FSM-specific—topic: chip overproduction. However, there exist methods that make use of a locked FSM to prevent chip overproduction. In this context, new FSM locking mechanisms were developed, which additionally add a Physical Unclonable Function (PUF) to generate device-specific locking keys, e.g. [AK07; AKP07]. Without the correct device-specific locking key, the functionality cannot be activated, which prevents overproduction. Such methods belong to the group of active hardware metering techniques.

The thesis does not further concentrate on this threat model.

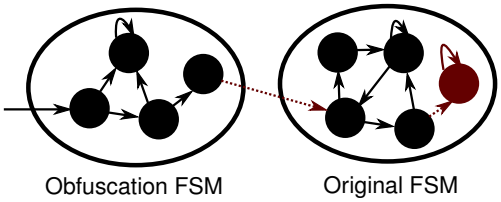
Reverse Engineering of Functionality (Second Threat Model): To extract the FSM's functionality, the attacker usually needs access to the gate-level netlist. This makes a successful attack more challenging for an end user than for a foundry because the end user first has to physically reverse engineer the product while the foundry has access to the netlist data without requiring RE, see Chapter 2.

Methods of the first threat model cannot necessarily be used for the second threat model. The changed conditions that hold for the second threat model can have a negative impact on the effectiveness of some of these locking-based methods of the first threat model [Kam+22], like for [CB09], [Des+13], or [DY18]. The work in [Fyr+18], for example, shows that RE can differentiate the original FSM from the obfuscation FSM, breaking the obfuscation method in [CB09], see Figure 5.1a. The obfuscation FSM corrupts the outputs and prevents the performance of the original FSM if a wrong input sequence is applied. A wrong input sequence hinders the correct traversal of the obfuscation FSM and thus the activation of the original functionality. However, RE reveals the structures of the obfuscation FSM and of the original FSM, which both form an SCC each and can thus be separated. Next, the secret, locking input pattern can be extracted by retracing the single available path that reaches the original FSM from the obfuscation FSM. As a result, the original FSM and the secret input sequence are identified, which breaks the locking mechanism. This attack is also discussed in [Mea+17] and improved in [Ros+21] and [Hu+21].

In contrast, other locking-based methods can also be used for the second threat model or were developed to consider the changed conditions for the second threat model. For instance, the works in [JS18; JS21a; JS21b; Mea+17; KJC19; LNO21; LO22; Pat+18] suggest to lock the original FSM itself (in addition to adding an activation mechanism), the work in [Rah+23] additionally strongly couples the obfuscation FSM and the original FSM, the work in [RB22] significantly increases the state space, or the works in [LO23; KMV20] use dynamically changing keys to lock the original FSM. The approach in [JS18; JS21a; JS21b] adapts the activation mechanism, using a hidden state transition to reach the original FSM, see Figure 5.1b. This hidden state transition can no longer be extracted by classical gate-level netlist functional RE. Instead, it is triggered by a timing glitch produced e.g. by a specific frequency change. To



(a) Harpoon [CB09]: Applying the correct input sequence to traverse the obfuscation FSM and reach the original FSM



(b) Hidden State Transitions (red dotted lines) [JS18; JS21a; JS21b]: Applying a timing glitch at the correct point in time to reach the original FSM, but do not end in the black hole state (highlighted in red)

Figure 5.1 Sequential obfuscation methods using activation mechanism with secret activation signal(s)

prevent the circumvention of the method by changing the frequency permanently, the method adds black hole states which are reached when the frequency is changed at the wrong point in time. Also, the work introduces time-dependent keys to modify the original FSM, what further increases the complexity of RE or Satisfiability (SAT)-based attacks [SRM15]. Karmakar et al. [KJC19] use the concept of (complemented) cellular automata to obfuscate FSMs. An interrupt logic translates an arbitrary FSM into a (complemented) cellular automata. For each FSM transition, a control signal, i.e. key input, decides if a cellular automata or a complemented cellular automata is used. Thus, a wrongly provided key sequence results in a wrong FSM. Similarly, for the approach in [LNO21; LO22], a control signal, i.e. key input, decides whether D-type or T-type FFs are used for the current transition of the obfuscated FSM. Thus, changing the key signal at the wrong moment results in a wrong FSM. Rahman et al. [Rah+23] combine different approaches to harden their technique against known attacks. They use both, input patterns and an external locking key, as secret to lock the correct FSM behavior. The locking key influences the output of an added counter, which encodes the FSM states, while the correct input pattern ensures the proper traversal of the added obfuscation FSM. The obfuscation FSM is strongly connected with the original FSM, what hardens the approach against structural attacks [Fyr+18]. Rahman and Bhunia [RB22] increase the possible state space of the original FSM what complicates RE significantly. For this, they include originally unused FSM states into the possible state space by reusing the sFFs for another circuit whose FFs also have FPs, but no unused states, e.g. a Nonlinear-Feedback Shift Register (NLFSR). The FF usage is controlled by a multiplexer, which is placed in front of each FF. Additionally, they add further sFFs that form an obfuscation FSM. For the correct locking input sequence, the obfuscation FSM reaches specific states that give the correct selector signals for the multiplexers. In [KMV20], a dynamic key is generated using a static, secret value and a Linear-Feedback Shift Register (LFSR) or a cellular automata. A trigger decides about when the key is changed. A key detection unit evaluates the correctness of the dynamic key and corrupts the sequential circuit and thus the FSM accordingly. In contrast, Li and Orailoglu [LO23] do not require an extra static, secret value to generate dynamically changing keys. The authors generate a signature for each FSM state. The signature represents the state transition sequence to reach this FSM state. To ensure that an FSM state receives the same signature calculation for every it reaching transition, they developed mechanisms to adapt the FSM accordingly. The generated signatures obfuscate the FSM states, which are automatically restored during runtime. Thus, the required key to restore the correct functionality is not static but changes for each FSM state, what hardens the technique against SAT-based attacks or RE attacks. However, as the key is part of the hardware design and is not inserted by a trusted party after production, the obfuscation approach does not protect against overproduction, see the first threat model.

Additionally, other methods were developed that prevent the second threat model and do not primarily rely on a functionality activation with a secret input pattern or key. The work in [Fyr+18] suggests a technique called Hardware Nanomites, which uses FPGA reconfiguration. The method divides the next state logic and output logic of the FSM into multiple separate parts. These parts are then configured on a dynamic physical block by the time needed. A register stores the last reached state while the dynamic physical block is reconfigured to describe the next part. Consequently, there is no point in time when the complete FSM design is physically available on the device. Thus, hardware sequential RE will fail. In return, the ASIC design must allow the integration of a dynamic physical block. For protection against the end user, there exists another promising new approach: Doppelganger [HP20]. Doppelganger strongly links FSM obfuscation with camouflaging. The method allows to choose a hidden and a visible functionality of the FSM. The visible functionality is implemented and synthesized. The hidden functionality can only be extracted if predefined gate connections are not considered. These gate connections are implemented as dummy gate inputs. If attackers cannot differentiate the real from the dummy inputs in a gate-level netlist, they obtain the visible instead of the hidden functionality. Thus, the designer can control the hidden functionality, i.e. the original control logic, and the visible functionality, i.e. the control logic which the attacker will assume to be the correct FSM. In return, the concept could also be misused to hide an HT when implementing the original FSM as visible functionality and the HT as hidden functionality.

Malicious Manipulation (Third Threat Model): The third threat model, the malicious manipulation of the FSM, is related to the second threat model. Suppose the functionality of the FSM is known to attackers. This can enable or ease malicious manipulations, e.g. for an effective HT or an effective fault injection [Qua+16], without requiring many random trials. So, in general, by preventing a successful extraction or RE of the FSM's functionality, such a target-oriented manipulation can be hindered or hardened.

In particular, for HT insertion or fault injection attacks, there also exist specific detection or blocking mechanisms. The goal of HT detection mechanisms is to verify if a product is HT free [e.g. HBS22] or to detect if an HT is currently active on a product [e.g. BGV15]. For fault injection attacks, detection mechanisms use, for example, redundancy [e.g. Bar+12], or special FSM encoding to avoid a malicious effect of state bit changes [e.g. CFT21]. These specific detection or blocking mechanisms are not further considered in this thesis.

This Thesis: The two novel obfuscation techniques presented in the following two sections use the second threat model and concentrate on the end user as the attacker. For the second threat model, FSM obfuscation methods that lock the functionality with secret keys, with dynamically changing keys, or based on input patterns may be used, e.g. [JS18; LO22; RB22; LO23]. Obfuscation schemes without a potentially attackable locking key are an alternative, like methods based on dynamic physical blocks [Fyr+18] or based on camouflaging techniques [HP20]. The two novel obfuscation techniques provide two further alternatives by hindering state-of-the-art RE tools to successfully identify the entire set of correct FSM gates in a gate-level netlist. The two novel obfuscation approaches are:

1. Timing Camouflage Enabled Finite State Machine Obfuscation ([Bru+22b], Section 5.2): Apply the camouflaging technique, Timing Camouflage, on sFFs to obfuscate FSMs.

2. Hardware Honeypots and Unattractive FSMs ([Bru+23], Section 5.3): Provide an attractive, misleading FSM while making the original FSM less attractive.

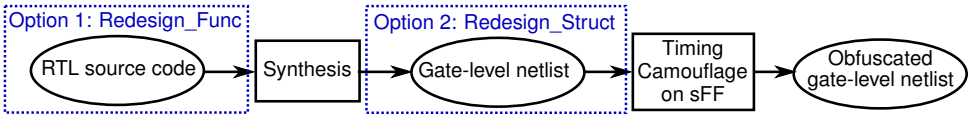


Figure 5.2 Workflow of Timing Camouflage Enabled FSM Obfuscation ©2022 IEEE [Bru+22b]

5.2 Timing Camouflage Enabled Finite State Machine Obfuscation

This section presents an FSM obfuscation idea, which is based on the camouflage method, Timing Camouflage (TC) [Zha+18b; Zha+20b]. TC removes FFs which are not required as long as the timing constraints of the affected paths and the correct clock frequency are maintained, see Section 5.2.1. We apply TC on FSMs, which turns out to be hard because sFFs often have combinational FPs (see Section 4.2.2) what prevents TC. However, if TC removes an sFF, RE algorithms, like RELIC [Mea+16] or REFSM [MZJ16], can no longer detect or use the removed sFF in the obfuscated netlist. As a result, the remaining sFFs lead to an obfuscated, extracted FSM. We present two methods to eliminate combinational FPs and so enable the removal of sFFs by TC. As shown in Figure 5.2, one of the methods, Redesign_Func, is applied at RTL code, whereas the other method, Redesign_Struct, is applied on the synthesized gate-level netlist. This section demonstrates that Timing Camouflage Enabled FSM Obfuscation (FSM-TC) can be a further step towards secure enough obfuscation of designs by offering an orthogonal method to other FSM obfuscation techniques.

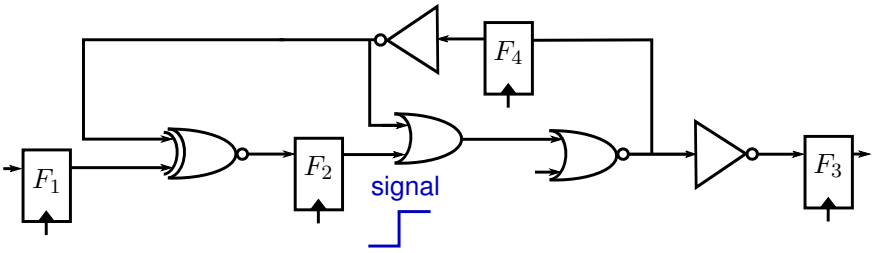
We consider the second threat model and an attacker, typically an end user, who has access to a reverse-engineered netlist without detailed timing information as is usually the case after optical layout extraction from a chip. We also assume that scan chains have been disabled after final chip testing.

5.2.1 Timing Camouflage

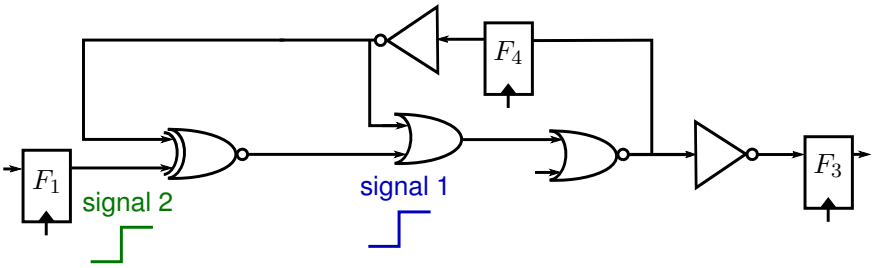
The following section introduces the camouflaging technique, called Timing Camouflage [Zha+18b; Zha+20b]. TC is based on a circuit optimization technique, called wave-pipelining [Bur+98; Cot69]. Wave-pipelining removes FFs from a gate-level netlist such that a path originally separated with one or multiple FFs now carries two or multiple data signals simultaneously. Thus, the clock period can be adapted, which can result in reduced circuit area, power, or latency. For a wave-pipelining path, the original functionality of the circuit can be maintained if the signal, which starts from an FF and propagates through the removed FF, reaches another FF after the first rising clock edge and before the second rising clock edge.

TC uses the concept of wave-pipelining as a camouflaging method to protect a gate-level netlist. Thus, TC also removes FFs from a gate-level netlist without changing the original circuit functionality. To ensure correct timing, and thus, a correct functionality, delays of gates and wires may need to be adjusted, or buffers may need to be added. As a result, the reverse-engineered, obfuscated netlist lacks the FFs which are removed by TC. An attacker who possesses the reverse-engineered, obfuscated netlist now requires the structure of the netlist and the timing information to extract the correct functionality of the circuit. If attackers assume that a wave-pipelining path carries only one data signal, RE will fail. For successful RE, they first need to identify the locations of the removed FFs, which complicates RE.

Figure 5.3 shows an example circuit (Figure 5.3a) and the corresponding camouflaged circuit (Figure 5.3b). For the original circuit, no wave-pipelining or TC is applied; thus, a signal from an



(a) Original circuit without wave-pipelining paths. Result: one signal per path.



(b) Obfuscated circuit with one wave-pipelining path by removing FF F_2. Result: two signals on wave-pipelining path, i.e. from F_1 to F_3 or F_4.

Figure 5.3 Concept of wave-pipelining and Timing Camouflage [Zha+20b] ©2022 IEEE [Bru+22b]

FF, e.g. F_2, reaches at another FF, e.g. F_3 or F_4, within one clock cycle. For the camouflaged circuit, wave-pipelining or TC is applied by removing the FF F_2 and adjusting the path delay. Now, a signal from F_2 reaches the next FF, i.e. F_3 or F_4, after one and before two clock cycles.

Besides timing constraints which make FFs more or less attractive for TC, there are FFs which cannot be removed directly: FFs with combinational FPs. If an FF with a combinational FP is removed, it will generate a combinational loop. This, however, violates digital design requirements because it introduces a high degree of instability and unreliability. For the example circuit in Figure 5.3a, F_4 has a combinational FP and can thus not be removed by TC.

Inspired by TC, other camouflaging techniques were developed, like TOIC [AGH19] or PhaseCamouflage [MS21]. TOIC hides the true timing behavior by using camouflaged sequential gates that implement a buffer, latch, or FF behavior. PhaseCamouflage is applied on adiabatic operations by obfuscating the phase difference of the power supply. Besides, there exist other locking techniques that involve circuit timing for their obfuscation technique, see Section 6.2.1.

Similar to other locking or camouflaging methods, the publication of TC led to the development of a new SAT attack to break TC, called TimingSAT attack [Li+18]. Paths whose delays can be estimated to be definitely within one clock period can be used to accelerate SAT attacks by excluding circuit nodes from further consideration. However, as elaborated in [Zha+20b], the complexity of the TimingSAT attack is significantly high in the case of large designs, no precise timing information, and added misleading paths, so-called false paths. On top of that, a deactivated scan chain increases the attack's complexity. Recently, another work extended the TimingSAT attack by identifying and removing false paths [MLS23].

5.2.2 Methodology

It is known that sFFs often have a combinational FP, see Section 4.2.2. However, this prevents the application of TC since a combinational loop will be created if an FF with a combinational FP is removed, see, for example, F_4 in Figure 5.3. As a consequence, we developed and implemented two techniques: *Redesign_Struct*, which modifies the gate-level netlist and

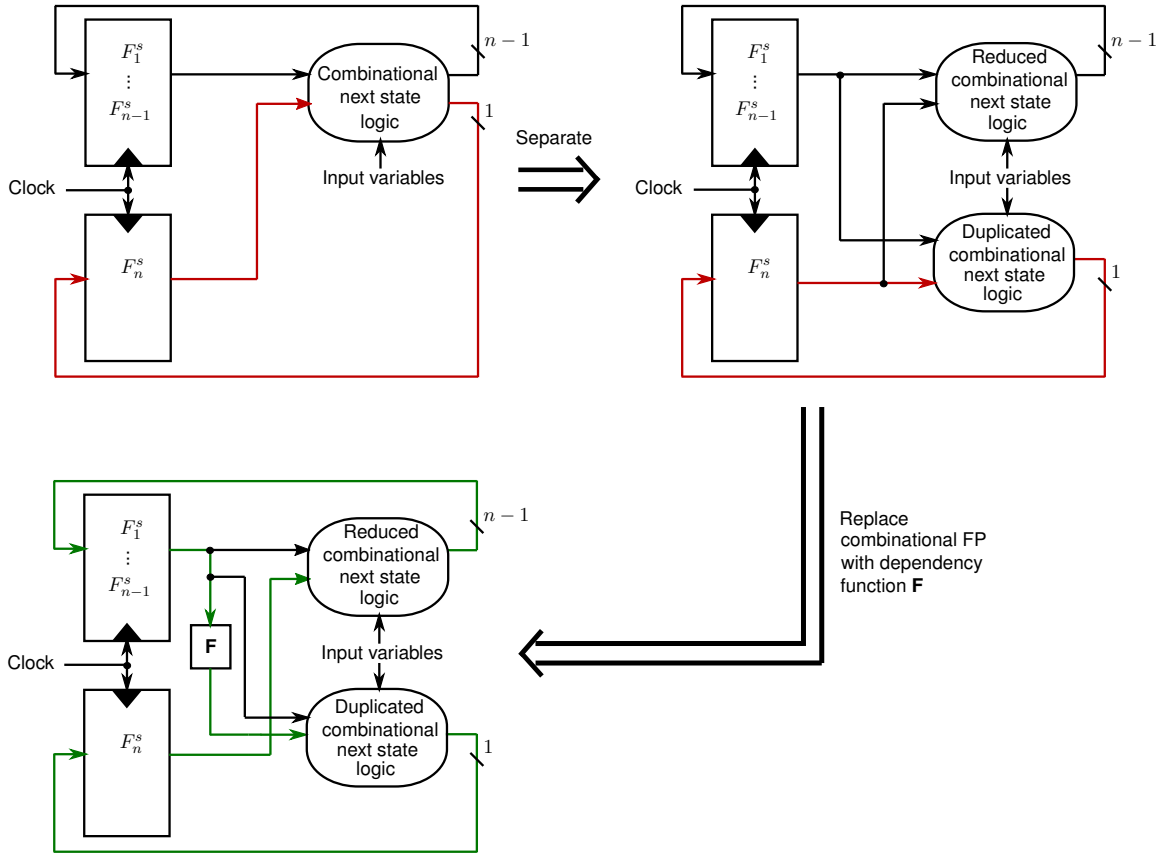


Figure 5.4 Redesign_Struct targeting F_n^s: separate combinational next state logic for F_n^s, and replace combinational FP (red) by introducing **F** ©2022 IEEE [Bru+22b]

requires a one-hot-encoded, targeted sFF; and *Redesign_Func*, which modifies the high-level description of an FSM, is applicable also for binary-encoded sFFs, but is more complex to implement (Figure 5.2). Both redesign an FSM such that at least one sFF will no longer contain a combinational FP but only FSM FPs, generic FPs, or no FPs, without changing the original functionality. This sFF—in the following assumed to be F_n^s—can then be removed by TC. Thus, we can achieve FSM-TC.

Redesign_Struct

Redesign_Struct modifies the gate-level netlist to ensure that one sFF does not have a combinational FP while the original functionality stays the same.

Methodology Assume an FSM with one-hot encoding. Thus, an sFF value $f_n^{s,t}$ at time t equals '1' if and only if all other sFF values, $f_1^{s,t}$ to $f_{n-1}^{s,t}$, equal '0'. This logic relation **F** determines $f_n^{s,t}$ without needing its previous value $f_n^{s,t-1}$, thus eliminating the combinational FP of F_n^s.

$$\textbf{F}: \quad f_n^{s,t} = \overline{(\vee_{j=1}^{n-1} f_j^{s,t})} \tag{5.1}$$

Redesign_Struct takes a gate-level netlist and the set of sFFs as inputs and outputs a modified gate-level netlist for a selected F_n^s. First, the tool separates the combinational FP of F_n^s from the rest of the combinational logic, see Figure 5.4. It duplicates the part of the next state logic which is driven by F_n^s and deletes gates of the original next state logic which

are solely driven by F_n^s. Finally, Redesign_Struct replaces the output of F_n^s which drives the combinational FP with the logic relation **F** in Equation (5.1). As a result, the combinational FP of F_n^s no longer exists, see Figure 5.4. To optimize the changes, the modified netlist is parsed to bench and synthesized again.

Encoding The relation **F** in Equation (5.1) is developed for one-hot-encoded FSM states, but also holds for otherwise encoded FSMs as long as at least F_n^s is one-hot encoded. To show an example, we evaluate the sFFs $\{F_3^s, F_2^s, F_1^s\}$ to the following FSM states: $\{100, 001, 010, 011\}$, resulting in **F**: $f_3^{s,t} = \overline{(\vee_{j=1}^{2} f_j^{s,t})}$.

Redesign_Func

Redesign_Func modifies the STT of the design's FSM to ensure that one sFF does not require a combinational FP after synthesis while the functionality stays the same.

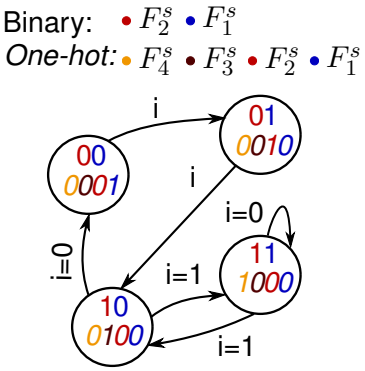
Methodology An sFF F_n^s has no combinational FP if a change to its next value $f_n^{s,t+1}$ is independent of its current value $f_n^{s,t}$:

Lemma 1 *An sFF F_n^s of $\mathcal{M}$ does not require a combinational FP if for each combination of k input values i_l^t, $l = \{k, \ldots, 1\}$ and $n-1$ other current sFF values $f_j^{s,t}$, $j = \{n-1, \ldots, 1\}$, there exist two transitions, α at time $t = t_1$ and β at time $t = t_2$, where the same combination forces $f_n^{s,t+1}$ either to '1' or to '0', independent of $f_n^{s,t}$:*

$$
\begin{aligned}
&\forall f_{n-1}^{s,t} \ldots f_1^{s,t} \times i_k^t \ldots i_1^t \text{ in } \mathcal{M}: \\
&\quad \exists \alpha, \exists \beta: \\
&\alpha: \quad f_n^{s,t_1} f_{n-1}^{s,t_1} \ldots f_1^{s,t_1} \times i_k^{t_1} \ldots i_1^{t_1} \rightarrow f_n^{s,t_1+1} f_{n-1}^{s,t_1+1} \ldots f_1^{s,t_1+1} \\
&\beta: \quad f_n^{s,t_2} f_{n-1}^{s,t_2} \ldots f_1^{s,t_2} \times i_k^{t_2} \ldots i_1^{t_2} \rightarrow f_n^{s,t_2+1} f_{n-1}^{s,t_2+1} \ldots f_1^{s,t_2+1} \\
&\textit{with} \quad f_n^{s,t_1} = \overline{f_n^{s,t_2}} \\
&\qquad\quad f_{n-1}^{s,t_1} \ldots f_1^{s,t_1} = f_{n-1}^{s,t_2} \ldots f_1^{s,t_2} \\
&\qquad\quad i_k^{t_1} \ldots i_1^{t_1} = i_k^{t_2} \ldots i_1^{t_2} \\
&\qquad\quad f_n^{s,t_1+1} = f_n^{s,t_2+1}
\end{aligned}
$$

The assumption for Lemma 1 is that all input variables I_l are either external inputs, FFs (initialized by reset), or wires without a combinational path from the targeted sFF F_n^s.

To achieve Lemma 1, one could rewrite the FSM and try to find a suitable implementation. This is, however, not a very promising solution. Instead, Redesign_Func adds artificial transitions and states to generate an FSM which fulfills Lemma 1, i.e. for each transition α there exists a transition β such that the conditions of Lemma 1 are fulfilled. Redesign_Func takes an STT and the FSM specific reset or error handling transitions as inputs and outputs a modified STT for a selected F_n^s. It starts with a transition pair check, which first adds all states to the FSM that are additionally required to satisfy the first two conditions for a transition pair in Lemma 1. All added states are supplied with the FSM specific reset or error handling transitions. Second, the transition pair check determines all suitable transition pairs according to Lemma 1 and a list L_f of all not paired transitions. Redesign_Func continues with the addition of obfuscation transitions. It adds transitions that are required to form new state transition pairs according to Lemma 1 with the transitions of L_f, see Algorithm 3. To preserve the original functionality, the

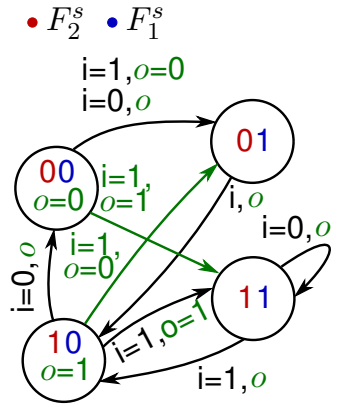


(a) Example FSM with binary and one-hot encoding

$f_2^{s,t}$	$f_1^{s,t} \times$ i		$f_2^{s,t+1}$	
0	0×0	$\rightarrow$	0	✓
1	0×0	$\rightarrow$	0	✓
0	1×0	$\rightarrow$	1	✓
1	1×0	$\rightarrow$	1	✓
0	1×1	$\rightarrow$	1	✓
1	1×1	$\rightarrow$	1	✓
0	0×1	$\rightarrow$	0	
1	0×1	$\rightarrow$	1	

(b) Transition pair check for binary-encoded FSM when targeting F_2^s

Figure 5.5 Original example FSM with four states and input value $i_1 = \text{i} = \{0, 1\}$. For simplification reasons and without loss of validity, the reset transitions to the reset state *00* are not considered here. ©2022 IEEE [Bru+22b]



(a) Example FSM with two artificial transitions, highlighted in green and controlled by obfuscation signal o

$f_2^{s,t}$	$f_1^{s,t} \times \text{i}\|\|o$		$f_2^{s,t+1}$	
0	$0 \times 0\|\|x$	$\rightarrow$	0	✓
1	$0 \times 0\|\|x$	$\rightarrow$	0	✓
0	$1 \times 0\|\|x$	$\rightarrow$	1	✓
1	$1 \times 0\|\|x$	$\rightarrow$	1	✓
0	$1 \times 1\|\|x$	$\rightarrow$	1	✓
1	$1 \times 1\|\|x$	$\rightarrow$	1	✓
0	$0 \times 1\|\|0$	$\rightarrow$	0	✓
1	$0 \times 1\|\|0$	$\rightarrow$	0	✓
1	$0 \times 1\|\|1$	$\rightarrow$	1	✓
0	$0 \times 1\|\|1$	$\rightarrow$	1	✓

(b) Transition pair check for modified FSM

Figure 5.6 Example FSM after Redesign_Func targeting F_2^s ©2022 IEEE [Bru+22b]

tool introduces an obfuscation signal o, which distinguishes between original and newly added artificial transitions. Whether $o = 0$ or $o = 1$ signals an original transition or not, depends on the current state. Note, if Algorithm 3 sorts the transitions in line 1 by their starting states, each state will have no (don't care) or exactly one correct signal o. *State dependent obfuscation signals* were also proposed in previous work, e.g. in [JS18] to introduce hidden and wrong state transitions. For all transitions that form already suitable transition pairs, the tool introduces two transitions, one with $o = 0$ and one with $o = 1$, see lines 2 and 3 in Algorithm 3. Finally, a second state transition check verifies that the modified STT now satisfies Lemma 1.

A small example FSM illustrates Redesign_Func, see Figure 5.5a. The table in Figure 5.5b checks Lemma 1 for F_2^s. The tool identifies a suitable transition pair (α, β) for all but two transitions, see the two rows on top of each other, e.g. rows 1 and 2, and the last column. Thus, the Lemma 1 does not hold for F_2^s. Also, in the synthesized netlist, F_2^s has a combinational FP, what entails an FSM redesign. Algorithm 3 adds two artificial transitions: $00 \times 11 \rightarrow 11$ and $10 \times 10 \rightarrow 01$, see Figure 5.6a. All other transitions stay the same and are extended by an obfuscation signal o. For state 00, the correct o is '0'; for state 10, it is '1'; for the two other states, it is a don't care value 'x'. Now, for all state transitions, a suitable state transition pair exists, see the table in Figure 5.6b. Thus, the Lemma 1 holds for F_2^s, i.e. there exists a netlist implementation without a combinational FP of F_2^s.

Algorithm 3 Pseudocode for generating a modified STT with original (♢) and added artificial (★) transitions by introducing an obfuscation signal o ©2022 IEEE [Bru+22b]

Data: STT T with transitions $\mathbb{E}(S_i, S_j)$, list of not paired transitions L_f, targeted sFF value f_n^s
Result: modified STT T_m

```
for e_l(S_i, S_j) in sorted transitions of T do
    if e_l(S_i, S_j) not in L_f then
        Add e_l(S_i, S_j)||o = 0 and e_l(S_i, S_j)||o = 1 to T_m        ▷ ♢
    else if e_l(S_i, S_j)||o = 1 not in T_m then
        Add e_l(S_i, S_j)||o = 0 to T_m                                 ▷ ♢
        y=((S_i with inverted f_n^s), e_l)
        e'_l(S'_i, S'_j) = e_l(S_i, S_j)
        for e_l(S_i, S_j) in transitions of T do
            if y=(S_i, e_l) then
                Add e_l(S_i, S_j)||o = 1 to T_m                         ▷ ♢
                Add e'_l(S'_i, S_j)||o = 1 to T_m                       ▷ ★
                Add e_l(S_i, S'_j)||o = 0 to T_m                        ▷ ★
                break
```

Handling of Obfuscation Signal Similar to other FSM obfuscation methods which are based on control signals dependent on FSM states or transitions, like [KJC19], [LNO21], or [LO23], the obfuscation signal o is a challenge and a chance for Redesign_Func. Often, authors suggest using an area-optimized list of transitions to match transitions with their correct, secret control signals [e.g. KJC19; LNO21], while the work in [LO23] proposes a dynamic key update function which calculates the correct control signals on the fly. In the following, we present three possibilities (Version A-C) to handle the obfuscation signal o for Redesign_Func. While Version A is applicable for all types of FSMs, Version B and C assume an FSM with a manageable number of options of how to run the FSM.

Version A: Extra state bit The signal o is defined as a one-bit register in the RTL source and assigned by the FSM. As a result, the sFFs are strongly connected to signal o. So, RE methods will likely identify it as additional sFF, increasing the FSM state labeling by one bit.

Version B: External input The obfuscation signal is defined as external one-bit input of the RTL source and applied by the end user of the designed chip. The designer therefore has to communicate the correct usage of this extra input, which depends on the previously applied inputs.

Version C: External locking key As with Version B, the obfuscation signal o is defined as external one-bit input of the RTL source. However, for Version C, the input is not applied directly by the end user. Instead, the designer has to provide a list of valid control signal sequences for the usage of the chip. The user applies a control signal sequence together with the list index, which selects the corresponding, correct sequence of o values from a secure device memory. The list index is communicated to the end user, while the corresponding o values stay secret and can therefore be used as locking keys.

Encoding and Extension Redesign_Func is designed for binary encoding. Other encodings are also applicable but can decrease the obfuscation success, see Section 5.2.3. We extend Redesign_Func to also accept don't care values (x-conditions) for the input variables I_l of STTs. This includes an extra pre-processing that resolves all those x-conditions into '0' and '1' values so that for each transition, a second transition with equal $f_{n-1}^{s,t}...f_1^{s,t}$ and $i_k^t...i_1^t$ exists (Lemma 1). Overall, this can improve the performance, see Section 5.2.3.

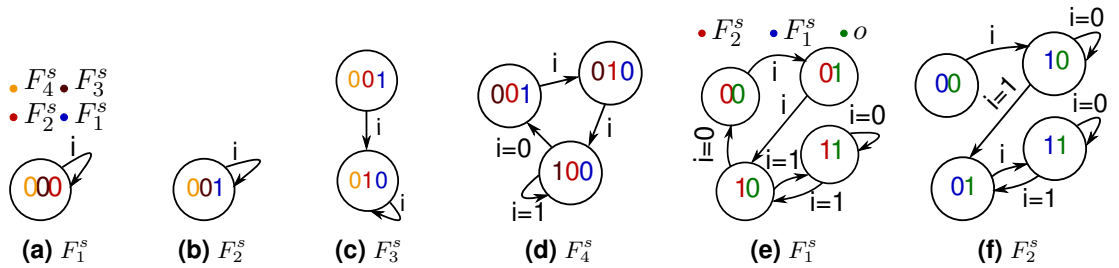


Figure 5.7 Extracted FSMs of example FSM in Figure 5.5a: (a)-(d) successfully obfuscated with FSM-TC (Redesign_Struct) while targeting sFFs F_1^s, F_2^s, F_3^s, and F_4^s; (e)-(f) obfuscated with FSM-TC (Redesign_Func, Version A) while targeting sFFs F_1^s and F_2^s ©2022 IEEE [Bru+22b]

Table 5.1 FSM extraction after FSM-TC (✓: successful obfuscation, x: obfuscation failed) ©2022 IEEE [Bru+22b]

example FSM	targeted sFF			
	F_1^s	F_2^s	F_3^s	F_4^s
Redesign_Struct	✓	✓	✓	✓
Redesign_Func	x[1]	✓	*)	*)

[1] predictable, and thus avoidable
*) not required for binary encoding

5.2.3 Results

The section demonstrates obfuscation results and overheads for FSM-TC.

Finite State Machine Extraction

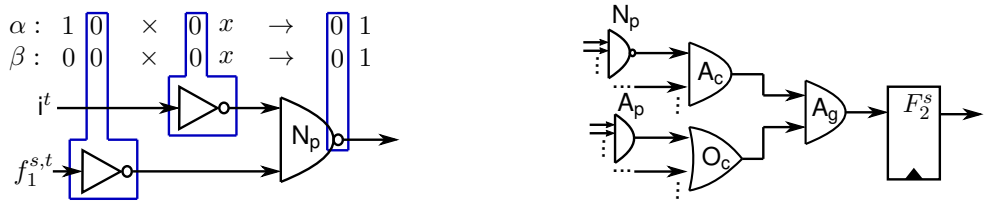
We synthesized using QFLOW [Ope19] and Yosys [Wol18] with default optimization settings and binary or one-hot FSM encoding. The tool *refsm* from the netlist analysis toolset NetA [MZJ19] extracts FSMs of not obfuscated and of obfuscated netlists to show their differences. The extraction starts with the reset state which the NetA tool *refsm* [MZJ19; Mea+19] identifies at first, and does not illustrate the reset behavior to increase the clarity of the extracted FSM. The results show the effect of the new obfuscation approaches on the FSM states and transitions.

Redesign_Struct To analyze the success of using Redesign_Struct, we obfuscate the *one-hot-encoded* example FSM in Figure 5.5a four times. Each time, we target a different sFF out of the four sFFs. The results show: all extracted FSMs are obfuscated, see Figures 5.7a-5.7d and Table 5.1. The obfuscation depends on the targeted sFF, e.g. for F_2^s, one state, while for F_4^s, three states were extracted.

To explain these results, we define the set $C' = \{c' | c'$ is a transition of the original FSM and assigns the next state value of the targeted sFF F_n^s to '1'$\}$, e.g. $0010 \times 0 \rightarrow 0100$ when targeting F_3^s. If F_n^s is removed, attackers experience an undefined FSM behavior when extracting c', e.g. $010 \times 0 \rightarrow 010$ in Figure 5.7c. Consequently, the sooner c' appears during the FSM extraction, the stronger the obfuscation might be.

Additionally, the selected, targeted sFF should influence at least one other sFF—often fulfilled for sFFs, see Section 4.2.2—to later enable an undefined FSM behavior for the obfuscation.

We also tested the approach on more realistic designs, a μC FSM (based on [Onc17b]) while targeting F_7^s and a Galois-counter mode AES core (based on [Ope10]) while targeting F_1^s, considering the assumptions of Lemma 1. For both, the synthesis re-encoded the FSMs to a one-hot encoding. The results also showed a wrong extraction of all visited c'.



(a) For $f_2^{s,t+1} = 0$, add inverted $f_1^{s,t}$ and inverted i^t to N_p

(b) Input all N_p to A_c, all A_p to O_c, and A_c and O_c to A_g which replaces input cone of F_2^s

Figure 5.8 Post-processing of synthesized netlist ©2022 IEEE [Bru+22b]

Redesign_Func To analyze the success of using Redesign_Func, we obfuscate the *binary-encoded* example FSM in Figure 5.5a twice, once by targeting the sFF F_1^s and once by targeting F_2^s.

The results for Version A show: the extracted FSM after targeting F_1^s is not obfuscated (compare Figure 5.5a and Figure 5.7e) and the extracted FSM after targeting F_2^s is obfuscated (compare Figure 5.5a and Figure 5.7f), see Table 5.1. Version B and C show an analogous behavior to Version A if choosing the correct external obfuscation signal o.

When targeting F_1^s, signal o takes over the role of the removed F_1^s because, for this example, Redesign_Func needs to add all transitions of one state as artificial transitions to another state. The extraction of the obfuscated design can now merge these two originally separated state labels into one without changing the possible transitions if o is chosen correctly. We can predict such behavior if the transition pair check cannot find any transition pair except for state-independent transitions, e.g. reset transitions. The appearance of such a case becomes more likely if Redesign_Func has to add many new states, as is increasingly the case for one-hot encoding. As a consequence, perform the transition pair check for each sFF once and choose the sFF which leads to the list L_f that contains the fewest transitions. For the example FSM, this is F_2^s with roughly 13% of transitions that fail the transition pair check, for the μC FSM this is F_3^s with roughly 16%, and for the AES core this is F_1^s with roughly 18%. Using these FFs for obfuscation resulted in obfuscated extracted FSMs.

Synthesis Post-Processing Redesign_Func is more prone to synthesis optimizations than Redesign_Struct. Its modified STT allows a synthesized netlist without a combinational FP for the targeted sFF F_n^s. However, for one STT, multiple valid netlist implementations exist that do not necessarily all satisfy this property. If the synthesis tool cannot be sufficiently controlled—as in our setup—it can also output an implementation with a combinational FP for F_n^s. So, we developed a post-processing to adapt the synthesized netlist (with synthesis option 'expose') if necessary. For each transition pair (α, β), the algorithm inverts the values of F_{n-1}^s to F_1^s and the input variables if they are equal to '0' in the current state, see Figure 5.8a. Next, for $f_n^{s,t+1} = 1$, the inverted and the not inverted signals (except x-conditions) are applied to an AND (A_p) gate; for $f_n^{s,t+1} = 0$, to a NAND (N_p). Then, the algorithm connects all A_p to another AND (A_c) and all N_p to an OR (O_c). Finally, A_c and O_c are connected to an AND gate (A_g) which replaces the input cone of F_n^s, see Figure 5.8b. Similar to Redesign_Struct, the adapted netlist is parsed and synthesized again; if necessary, now also with reduced optimization settings.

Security Discussion In contrast to most FSM obfuscations, FSM-TC (Version A and B) has no secret locking key or input sequence which could be exploited by typical attacks, see Section 5.1. Instead, attackers need to identify the locations of the removed sFFs, which is assumed to be difficult if, in addition to FSM-TC, also standard TC is applied on the complete, and not only on the FSM circuit (Section 5.2.1). Besides, to attack Redesign_Struct, attackers could repeat the FSM extraction several times and, each time, choose a not yet extracted state as

Table 5.2 Comparison of STTs without and with x-conditions: number of transitions, and average runtime of Redesign_Func without and with pre-processing ©2022 IEEE [Bru+22b]

Design	without x-conditions		with x-conditions	
	#trans.	time [s]	#trans.	time [s]
example FSM	16	0.006	10	0.008
μC FSM	896	0.398	29	0.012
AES core	20480	343.628	36	0.019

the starting state. Usually, FSM extraction tools, like *refsm*, use the FSM's reset state as the starting state, which they identify in a first step. However, an attacker could repeat the FSM extraction using the same sFF set but using a different starting state, which does not appear in the already extracted FSM. As a result, a different FSM will be extracted that contains solely new FSM states or a mixture of new and old FSM states. Finally, a subset of all extracted FSMs might equal the original FSM, however, without correct transitions $c' \in C'$. The more reasonable, new starting states exist, e.g. for a mixed encoding (Section 5.2.2), the more difficult it becomes for attackers to identify this subset of FSMs and the wrong transitions.

Overhead of Finite State Machine Redesign

This section discusses the consumed runtime and the cell area overhead of the FSM redesign methodology.

We measured the runtime of Redesign_Func with and without pre-processing, so with and without x-conditions in the STTs (Section 5.2.2). STTs with x-conditions usually have significantly fewer transitions than ones without (Table 5.2). The table also shows the average runtime of three designs by targeting each sFF once. For designs with small differences in the number of transitions, the runtime did not decrease (example FSM). However, for designs with large differences, the runtime decreased significantly (AES core).

We measured the cell area of the redesigned netlists with a proprietary Electronic Design Automation (EDA) tool. For each design, we chose one targeted sFF. On average, the area overhead was 15% for Version A of Redesign_Func and only 2% for Version B or C because Version B or C does not need to determine o by an extra circuit logic. Version C requires additional area for the secure memory. For the investigated designs, Redesign_Struct decreased the average area consumption by 12%. Besides the FSM redesign, also TC will increase area, see [Zha+20b]. Finally, an FSM is usually part of a bigger design what decreases the impact of the area overhead of FSM redesign on the overall area consumption.

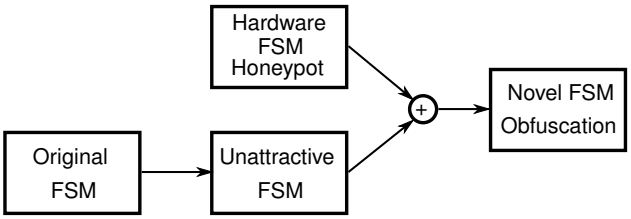


Figure 5.9 Novel two-part FSM obfuscation approach: hiding the original FSM by making it less attractive (unattractive FSM) and providing an attractive alternative in form of a hardware FSM honeypot

5.3 Hardware Honeypots and Unattractive Finite State Machines

FSM-TC in Section 5.2 and Doppelganger in [HP20] use the key-less method, camouflaging, to achieve obfuscation. Camouflaging, however, often requires a foundry with special capabilities to implement it into the design, like adding a thin isolating layer to gate contacts [HP20]; or the foundry is able to reveal the camouflaged information, like the gate and wire delays for FSM-TC in Section 5.2, without great effort. This section, in contrast, presents a new technique that is not based on foundry-enabled camouflaging or on key-based locking. It hinders state-of-the-art RE methods from successfully identifying the entire set of correct FSM gates in a gate-level netlist by purposefully exploiting characteristics of RE methods, which results in a lack of necessary information, similar to camouflaging. In addition and similar to [HP20], it leads the attacker to a wrong, designer-controlled FSM.

Many state-of-the-art sequential RE methods do not fully investigate the extraction of multiple FSMs [Kib+22]. There are methods that extract multiple FSM candidates but do not further elaborate on how to choose the correct FSMs out of multiple FSM candidates, like the method in [Shi+10] or the post-processing technique in Section 4.1.2. Other methods extract only one FSM candidate, e.g. fastRELIC [BBS19] or ReIGNN [CYN21]. Thus, the existence of multiple FSMs within a design complicates sequential RE. In addition, current sFF identification methods are heuristic approaches that use specific—often similar—features to identify sFFs.

We take advantage of these two properties and present a novel two-part FSM obfuscation methodology to prevent sequential RE, see Figure 5.9.

- We introduce *hardware Finite State Machine Honeypots (FSM-HPs)* which satisfy features of the sFF identification methods. FSM-HPs pretend to be the correct FSM of the design, which causes attackers to stop their effort to extract further FSMs and thus prevents the extraction of the correct FSM. To ensure that state-of-the-art sequential RE methods identify the FSM-HP as a single FSM or as the best suitable candidate, we design the FSM-HP to be more attractive than the correct FSM.

- We obfuscate the original FSMs, now called *unattractive FSMs*, by eliminating features of certain sFF identification methods. As a result, unattractive FSMs are resistant to these FSM identification methods. Similar approaches exist, for example, in the context of HT insertion [Che+22] or logic locking [CYN23], which insert HTs or logic locking gates with weaker features of their detection or breaking methods to circumvent these attacks.

- We combine both techniques, FSM-HPs and unattractive FSMs, to enhance the effect of both.

FSM-HPs can be implemented on RTL or on gate level, allowing the designer to freely control design properties or design functionality. This allows them to increase the attractiveness of

the FSM-HP and engage an attacker with controlled, false information. The results show that by using our novel FSM obfuscation methodology, state-of-the-art sFF identification methods favor the sFFs of the FSM-HPs or can no longer identify the correct sFFs. Both lead to a wrong FSM extraction. Our threat model assumes access to a reverse-engineered gate-level netlist and sequential RE tools but no access to further design specifications, like simulation stimuli. Thus, since attackers do not know the correct FSM, they cannot expose an extracted, plausible-looking FSM-HP as fake FSM, even if they are aware of FSM-HPs.

5.3.1 Exploitable Finite State Machine Extraction Features

To design a highly attractive FSM-HP for state-of-the-art sFF identification methods, it should meet as many of the identification methods' features as possible. If not all features can be achieved for an FSM-HP design, one can choose features that are frequently used by state-of-the-art sFF identification methods. We identified these frequently used features by analyzing and comparing state-of-the-art sFF identification methods in Section 3.5.3. The results in Table 3.1 give us the following three features: the combinational FP, the dissimilarity, and the influence/dependency behavior. Thus, these features form good target features when designing attractive FSM-HPs.

In addition, we identify two features that have a significant impact on the success of identification methods and can be avoided during FSM design: combinational FP and dissimilarity (highlighted in Table 3.1). We show that these two features can be exploited to build unattractive FSMs. We change FSM designs such that not all of their sFFs possess all of these features without changing their original functionality. As a result, sFF identification methods that use these features will not correctly identify all sFFs, and thus, a correct RE of unattractive FSMs will fail.

Also, adapting sFF identification approaches to better identify unattractive FSMs is not assumed to be a promising solution. If identification methods would use less restrictive features, such that they also identify sFFs of unattractive FSMs, the false positive rate, i.e. the number of FFs which are wrongly identified as sFFs, is expected to drastically increase.

5.3.2 Methodology

The following section introduces the two parts of the new FSM obfuscation methodology: hardware FSM-HPs and unattractive FSMs.

Hardware Finite State Machine Honeypot

The first part of the new FSM obfuscation methodology is hardware FSM-HPs, which pretend to be the correct FSMs. These FSM-HPs must be more attractive for sequential RE methods than the original FSMs so that they are identified as best FSM candidates. Additionally, FSM-HPs only pretend to impact the design's functionality but must not change the functionality in reality. We assume that no further limitations exist for FSM-HPs.

An FSM-HP will be added to the original design, e.g. as a separate module. To avoid easy detection due to its isolated, unconnected appearance, we use original design inputs as inputs for the FSM-HP, in particular, the original design's reset and clock signal. The outputs of the FSM-HP should pretend to control the design behavior, e.g. by using techniques like dummy contacts [CBC07], logic redundancy [LV62], or primary module outputs. To strengthen FSM-HPs not only against structural but also against functional detection techniques, FSM-HPs should pretend to process data if the netlist is, for example, simulated. This can, for instance, be achieved with the above-described techniques of how to connect FSM-HPs' inputs and

outputs. Additionally, the more of the typical FSM features an FSM-HP fulfills—like the ones in Table 3.1—the more attractive it becomes for state-of-the-art RE methods. Two highly relevant features are combinational FPs and a firm influence/dependency behavior, e.g. that the sFFs of the FSM-HP belong to the same SCC. Both features can be easily verified for a gate-level netlist representation of an FSM-HP design. Section 5.3.3 presents two concrete FSM-HP implementations. However, the work aims to show the variety of design options rather than to give plenty of individual design instructions.

Unattractive Finite State Machines

The second part of the new FSM obfuscation methodology is unattractive FSMs, which help to make FSM-HPs more attractive than the original FSMs. An unattractive FSM has a special design that exploits a feature of one or more specific sFF identification algorithms, see Section 5.3.1. By designing an FSM so that it does explicitly not fulfill a specific sFF feature, which is a feature for an sFF identification algorithm, the algorithm and thus the FSM extraction fails. There exist different strategies to achieve such an FSM design. Either the designer is aware of the requirements and designs the FSM accordingly, or an existing FSM is redesigned without changing the original functionality. If possible, the second strategy is preferred, as it can be done independently of the FSM design process. In the following, we introduce two redesign methods to build an unattractive FSM, based on the identified features in Section 3.5.3 and 5.3.1: dissimilarity and combinational FP.

Dissimilarity Approach The dissimilarity feature makes use of the fact that the input structure of sFFs is usually less similar than the input structure of data FFs because due to data words, data bits are often processed in a similar way [Mea+16]. An unattractive FSM should have a low dissimilarity and, thus, a high similarity score. One can calculate a similarity score for an FF by comparing its FF input structure with all other FF input structures of the design [Mea+16]. To increase the similarity score of sFF input structures, we replicate each state bit in the RTL description multiple times. As an example, assume an FSM with three state bits and the following six states $\{S_1, \ldots, S_6\}$:

$$S_1 = 000, S_2 = 001, S_3 = 010, S_4 = 011, S_5 = 100, S_6 = 101$$

After replicating each state bit twice (marked in blue), the six states have the following labels:

$$S_1 = 000\,000\,000, S_2 = 000\,000\,111, S_3 = 000\,111\,000,$$
$$S_4 = 000\,111\,111, S_5 = 111\,000\,000, S_6 = 111\,000\,111$$

The synthesis options are modified so that no re-encoding of the FSM and no merging of the FFs occur and, thus, the replicated state bits are translated into individual FFs. State bit replication increases the number of required sFFs, including their input logic, but does not change the overall design functionality, as only the state labeling is changed. As a result, all FFs representing the replicated bits of one state bit will have a highly similar input structure that increases the overall similarity score of these sFFs and makes them more difficult to identify as sFFs.

In the specific case of RELIC-Tarjan [Mea+19; MZJ19], it may not be sufficient to solely increase the similarity of sFFs, i.e. solely decrease the Z-Score values of sFF signals because RELIC-Tarjan uses only one signal with the highest Z-Score value to identify the corresponding set of sFFs (see Section 3.5.2). As a consequence, the identification will also succeed if any FF of the FSM SCC—even if it is not an sFF—has the highest Z-Score value.

To show an example, assume an FSM has two sFF signals, F_1^s and F_2^s, which belong to the same FSM SCC, scc, and scc additionally contains three other, nFF signals, F_1^n, F_2^n, and F_3^n:

$$scc : \{F_1^s, F_2^s, F_1^n, F_2^n, F_3^n\}$$

Assume that before applying the dissimilarity approach, RELIC-Tarjan determines the highest Z-Score value to be 622 and that it belongs to F_2^s:

$$scc : \left\{ \begin{array}{ll} F_1^s : & \text{Z-Score} = 512 \\ \boldsymbol{F_2^s :} & \textbf{Z-Score} = \mathbf{622} \\ F_1^n : & \text{Z-Score} = 84 \\ F_2^n : & \text{Z-Score } = 389 \\ F_3^n : & \text{Z-Score} = 110 \end{array} \right\}$$

Thus, all FF signals in scc will be identified as sFFs, including the correct sFF signals, F_1^s and F_2^s. After applying the dissimilarity approach on all sFFs, the Z-Score values of F_1^s and F_2^s decrease. However, it might happen that now the highest Z-Score value is 389 which again belongs to one of the FF signals in scc, namely F_2^n:

$$scc : \left\{ \begin{array}{ll} F_1^s : & \text{Z-Score} = 178 \\ F_2^s : & \text{Z-Score} = 209 \\ F_1^n : & \text{Z-Score} = 84 \\ \boldsymbol{F_2^n :} & \textbf{Z-Score} = \mathbf{389} \\ F_3^n : & \text{Z-Score} = 110 \end{array} \right\}$$

Consequently, again, all FF signals in scc—including F_1^s and F_2^s—are classified as sFFs, which hinders a successful obfuscation.

To prevent this, one could replicate all signals from the FSM SCC; however, for most cases, replicating state and counter bits is sufficient because they often have the highest Z-Score values. Counter bit replication appears to be more challenging than state bit replication because counters usually have significantly more states than FSMs, e.g. 256 states for an 8-bit counter. Furthermore, in contrast to states, counters are usually not assigned within a case structure but by assignments that count up or down. We developed a technique that allows a counter bit replication without using a costly case structure for the counter state assignment. Instead of introducing a new counter for each bit replication, we increase the number of bits of the existing counter, like it was done for FSM states. To maintain the counter behavior, we increment or decrement an extra counter signal and then assign the replicated bits to the same value of the original counter bit. This method is valid as long as no counter over- or underflow occurs. As an example, assume a 3-bit counter register `c`, which counts up as shown in Table 5.3 using the following source code in the original design:

```
c <= c + 1;
```

We replicate each counter bit twice by adding a 9-bit temporary `c_t` and following code lines:

```
c_t = c + 1;
c[8:6] <= (c_t[8:6]==3'b001)?3'b111:c_t[8:6];
c[5:3] <= (c_t[5:3]==3'b001)?3'b111:c_t[5:3];
c[2:0] <= (c_t[2:0]==3'b001)?3'b111:c_t[2:0];
```

With this method, similar to the FSM state bit replication, the design's functionality is not affected, as only the counter state labeling is changed, e.g. the original counter state 001 is now labeled as $000\,000\,111$, see Table 5.3. As a result, the similarity score of the FFs belonging to the counter will increase, hindering the identification of the correct FSM SCC.

Table 5.3 Counter assignment for an exemplary 3-bit counter `c` before the counter bit replication and for the 9-bit (temporary) counters `c_t` and `c` after each counter bit is replicated twice.

	original	obfuscated	
	`c`	`c_t`	`c`
	000	000 000 000	000 000 000
+1	001	000 000 001	000 000 111
+1	010	000 001 000	000 111 000
+1	011	000 111 001	000 111 111
+1	100	001 000 000	111 000 000
+1	101	111 000 001	111 000 111
+1	110	111 001 000	111 111 000
+1	111	111 111 001	111 111 111

Feedback Path Approach The combinational FP feature makes use of the fact that sFFs often have a combinational FP (see Section 4.2.2). This feature was one of the first features used for sFF identification [McE01; Shi+10] and is still used in recent approaches [Fyr+18].

FSM-TC in Section 5.2 requires sFFs without combinational FPs to apply Timing Camouflage. For this, we developed two methods to avoid a combinational FP for an sFF without changing its original functionality by redesigning an FSM (Section 5.2.2): Redesign_Struct which is applied on one-hot-encoded FSMs and on gate-level netlists, and Redesign_Func which is applied on binary-encoded FSMs and on RTL code. Redesign_Struct partially disconnects an sFF from posterior logic in such a way that the combinational FP is removed, see the example in Figure 5.4. Where disconnected, the signal is replaced by a Boolean function **F** that outputs one if all other sFF values equal zero—the definition of one-hot encoding. Redesign_Func adds extra, dummy transitions to the RTL description of the FSM and controls them by an obfuscation signal, see the example in Figure 5.5a and Figure 5.6a. The dummy transitions are designed such that one of the FSM bits can always be determined without considering its own value, i.e. using only the other FSM bit values and inputs. As a result, both methods ensure that at least one of the sFFs of an FSM does not influence itself within one clock cycle. Consequently, this sFF is free of combinational FPs while the original FSM functionality does not change.

The FP approach adopts these techniques by using the resulting, redesigned FSM as unattractive FSM. sFF identification methods which use combinational FPs as feature will no longer identify this sFF with FSM or generic FP. This results in an unsuccessful sFF identification and, consequently, in an unsuccessful FSM extraction.

Design Options

The design options and complexities to generate unattractive FSMs are defined by the applied redesign method. While the dissimilarity approach is implemented by hand, the redesign method of the FP approach can be partly automated and was shown to have short runtimes, see the results in Section 5.2.3.

In contrast to unattractive FSMs, there are various options on how to design an FSM-HP. They can be built on RTL level or as gate-level netlists, by hand or automatically by a generator, with identical or changed synthesis options. Each design option has different advantages and disadvantages. Designing on RTL level or by hand allows for an FSM-HP with a user-defined functionality. This enables FSM-HPs which lead the attacker to targeted wrong conclusions about the design. Designing on a gate-level netlist enables better control over the final netlist structure because the synthesis will not determine the gate representation itself. Better control

can ease the achievement of attractive gate-based features, like dissimilarity. If the FSM-HP is designed automatically by a generator, one can create a high number of FSM-HPs variations in a short time. By providing different parameters or adding specific conditions to the generator, the designed FSM-HPs can be forced to satisfy predefined features, like the number of state bits or combinational FPs. Separate synthesis processes for the original FSM and the FSM-HP are also possible. This allows the designer to maintain all design-specific synthesis settings for the original design while choosing suitable settings for the FSM-HP to achieve maximum attractiveness.

The complexity of generating an FSM-HP is independent of the remaining design. Thus, the complexity does not change whether the FSM-HP is generated for a toy example or for an industrial design. When using automatic FSM-HP generators, instead of generating it by hand, the runtime to build an FSM-HP can be negligibly small.

5.3.3 Results

We demonstrate the different obfuscation approaches using nine open-source designs, including designs from OpenCores (*aes_core* [Ope16a], *altor32_lite* [Ope15], *fpSqrt* [Ope18], *gcm_aes* [Ope10], a *uart* based on [Ope16b]), cryptography designs (*sha1_core* [Str18b], *siphash* [Str16]), and a submodule as well as a complete core of a RISC-V processor (*mem_interface* [Onc17b], *picorv32* [Yos22]). For synthesis, the open-source tools QFLOW [Ope19] and Yosys [Wol18] are applied without adding specific timing constraints. Table 5.4 provides additional information on the designs and their synthesized netlists: the number of FFs, of SCCs with at least two FFs, of FSMs, of sFFs per FSM, and the encoding of the FSM states after synthesis. The synthesis setup adjusts all identified FSMs, resulting in a one-hot encoding for most of the designs. Three designs, *uart*, *mem_int.*, and *picorv32*, consist of only a single source code module, while all others are composed of a minimum of two modules. The use of the 32-bit RISC-V core demonstrates the adaption for realistic designs.

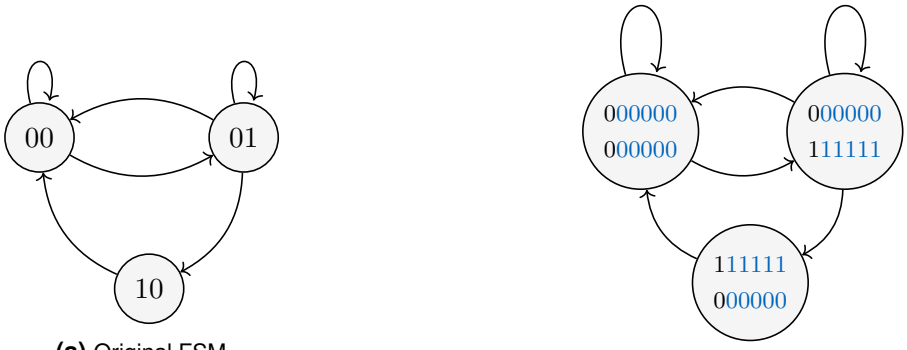
We evaluate the obfuscation results using two sFF identification approaches: RELIC-Tarjan [MZJ19] and the topological analysis [Fyr+18], see Section 3.5.2. To assess the success of the obfuscation technique, we compare the success of identifying the sFFs in the original and in the obfuscated netlist. For the obfuscated netlists, we additionally differ between the successful identification of sFFs of the unattractive FSM and the successful identification of sFFs of the FSM-HP. The sFF identification is considered to be successful if 100% sensitivity is achieved, i.e. if all sFFs are part of the identified sFF set. It does not take into account the amount of nFFs which are wrongly identified as sFFs because the obfuscation technique aims a high or perfect sensitivity when identifying the sFFs of the FSM-HP and a low or zero sensitivity when identifying the sFFs of the unattractive FSM, but has no aim regarding the nFFs.

RELIC-Tarjan

This section shows the successful obfuscation, using the dissimilarity approach combined with an FSM-HP, against similarity-based sFF identification. For each design, we synthesize the original and the obfuscated version. As explained for the dissimilarity approach in Section 5.3.2, the obfuscated designs are synthesized without FSM re-encoding and FF merging to preserve the sFF replication on gate-level. For better comparability, we also deactivate the re-encoding when synthesizing the original netlists. As a result, some FSMs of Table 5.4 remain binary-encoded instead of being changed to one-hot encoding. We then evaluate the obfuscation by applying the RELIC-Tarjan approach on both netlists, using the default settings and the option *buf* for the tool *relic*. We make the same small adjustment for RELIC-Tarjan as we did for the evaluation of the post-processing methods in Section 4.2.4: The FF signal with

Table 5.4 Single and multi-module benchmarks and their number of FFs, SCCs with at least two FFs, FSMs, sFFs and the type of encoding when synthesizing with default optimization settings ©2024 IEEE

Design	#FFs	#SCCs	#FSMs	#sFFs (per FSM)	encoding
uart	64	2	2	3,2	binary
mem_int.	75	1	1	7	one-hot
siphash	794	2	1	8	one-hot
sha1_core	850	3	1	3	one-hot
aes_core	901	2	1	16	one-hot
altor32_l	1249	2	1	6	one-hot
fpSqrt	1331	2	1	3	one-hot
gcm_aes	1697	5	1	10	one-hot
picorv32	1598	1	2	4,7	one-hot



(a) Original FSM

(b) Unattractive FSM

Figure 5.10 FSM encoding of the design *fpSqrt* before and after replicating its state bits five times ©2024 IEEE

the highest Z-score value chooses the SCC which must contain at least two, instead of one, FF signals as elements, because it is assumed that an FSM with only a single sFF is very unlikely, see also the topological analysis in [Fyr+18]. Thus, when using example designs that contain only one SCC with at least two FFs, RELIC-Tarjan will always output this SCC. Similarly, when using example designs that contain only FSM SCCs and no data SCCs, RELIC-Tarjan will always output an FSM SCC. So, to achieve a more significant evaluation, we use designs with one FSM and which have at least two SCCs with more than one FF, i.e. designs *siphash* to *gcm_aes*, see Table 5.4. For the following analysis, besides the FSM SCC and the data SCCs, we label another SCC specifically: the FSM-HP SCC contains all or the majority of sFFs of the FSM-HP. Similar to the FSM SCC, also the FSM-HP SCC might contain other FFs in addition to the sFFs of the FSM-HP.

For the dissimilarity approach, we replicate state bits and, if necessary, other signals of the FSM SCC three, five, or 31 times, depending on the achieved Z-Score reduction. The example in Figure 5.10 shows the FSM of the design *fpSqrt* before and after the FSM state bit replication. To design an FSM-HP, we copy the original FSM source code, modify its next state or output logic, and use its outputs as primary module outputs. This keeps realistic FSM features. For some designs, this process had to be repeated to receive an FSM-HP SCC element with a Z-Score value higher than those of the FSM SCC. We recognized an increased challenge to find such a suitable FSM-HP if the original FSM either has few state bits, e.g. two, or has a strong cyclic behavior. We assume that the sFFs of such FSMs have features besides

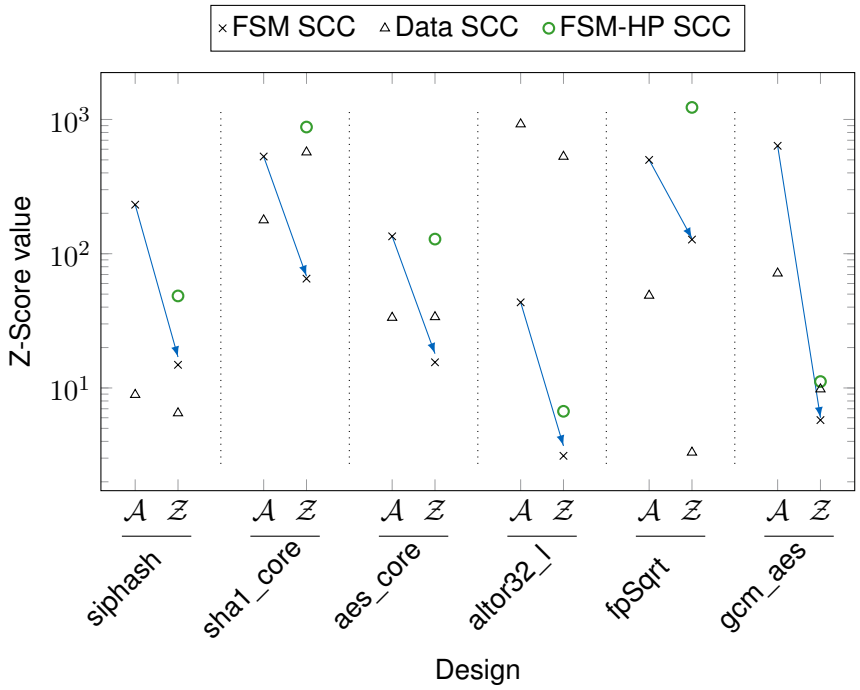


Figure 5.11 Maximum Z-Score value of the FFs in any data SCC and of the FFs in the FSM SCC before ($\mathcal{A}$) and after ($\mathcal{Z}$) obfuscation, and maximum Z-Score value of the FFs in the FSM-HP SCC ©2024 IEEE

the similarity score that are well identifiable by the tool *relic*. Note that it is realistic to assume that designers are able to evaluate their FSM-HP or unattractive FSM design, e.g. regarding Z-Score values, to decide about further design adaptions: a designer also has access to RE tools and can thus apply them.

Figure 5.11 plots the maximum Z-Score value of the FFs in any data SCC and the maximum Z-Score of the FFs in the FSM SCC using the original netlists ($\mathcal{A}$), compared against the maximum Z-Score value of the FFs in any data SCC, the maximum Z-Score value of the FFs in the FSM SCC, and the maximum Z-Score of the FFs in the FSM-HP SCC using the obfuscated netlists ($\mathcal{Z}$). The figure shows that the obfuscation succeeds for all designs, as an FSM-HP could always be designed such that at least one FF in its SCC has a higher Z-Score value than any FF of the FSM SCC (compare the green circles and the black crosses for the obfuscated netlists $\mathcal{Z}$). Due to the dissimilarity approach, for all designs, the maximum Z-Score value of the FFs in an FSM SCC decreased for the obfuscated design. We highlight this change with blue arrows. For the majority of designs, we achieve a maximum Z-Score value for the FFs of the original FSM SCC which is below the maximum Z-Scores for the FFs of data SCCs and the FSM-HP SCC. Note that the given Z-Score values in Figure 5.11 represent only the maximum value of an FF within the FSM SCC, the FSM-HP SCC, and all data SCCs. The Z-Score values of all remaining FFs, including the maximum Z-Score values of further data SCCs, are not visible in Figure 5.11. Their Z-Score values would result in further plot points below the black cross, the black triangle, or the green circle. In summary, the RELIC-Tarjan procedure will favor the FSM-HP over the unattractive FSM as the best FSM candidate. Thus, the sFFs of the FSM-HP are successfully identified (perfect sensitivity), while the sFFs of the unattractive FSM are not successfully identified (zero sensitivity). This allows a successful FSM obfuscation.

The following briefly discusses possible attack scenarios for this evaluation experiment.

- The ranking position of the maximum Z-Score value of the FSM SCC should not be predictable to hinder attacks, like selecting the SCC with the smallest instead of the

Table 5.5 Identifying sFFs with topological analysis with and without obfuscation (✓: all correct sFFs identified, (x/y): x out of y sFFs correctly identified resulting in a successful obfuscation)

Design	without obfuscation	with FP approach and FSM-HP	
	orig. FSM	orig. FSM	FSM-HP
uart	✓	(1/2), (2/3)	✓
mem_int.	✓	(0/7)	✓
siphash	✓	(7/8)	✓
picorv32 [1]	✓	(3/4)	✓

[1] FSM of the memory interface

highest maximum Z-Score value. Adding more than one FSM-HP to the design supports a not predictable ranking and, thus, strengthens the obfuscation.

- If an attacker would extract more than one FSM candidate, the attack's complexity increases linearly with each additional extraction. Nevertheless, after extraction, the attacker has to distinguish the correct FSM from plausible-looking FSM-HPs. To complicate the identification of unattractive FSMs, the designer can add additional FSM-HPs which also possess an altered state labeling, e.g. with replicated state bits.

- If an attacker is aware of the dissimilarity approach, he/she could try to identify and remove the replicated logic, e.g. by resynthesis. This would weaken the obfuscation, as the attractiveness of the FSM-HP would no longer be supported by an unattractive original FSM; plus, the removed logic could give the attacker a hint where to find the original FSM. Nevertheless, if the FSM-HP is sufficiently attractive and the removed redundant logic cannot be uniquely mapped to the original FSM, e.g. by adding additional FSM-HPs with replicated state bits, the obfuscation is still valid. To complicate the removal attack, the designer can add countermeasures, like key-controlled logic [RKM08a] or dummy contacts [CBC07], to the replicated logic. Also, if the sFF's input cone is not perfectly replicated but with a small adaption, this might hinder removal attacks. Such an adaption is assumed to have no significant impact on the input cones' similarities if it appears at a sufficient distance to the sFF's input.

Topological Analysis

The following section shows the successful obfuscation using the FP approach combined with an FSM-HP against topological-analysis-based sFF identification. We use four designs of Table 5.4 for which the topological analysis is able to extract an FSM candidate with all sFFs of the original FSM, see the second column of Table 5.5. As discussed in Section 3.5.2, the last step of the topological analysis is the determination of the control behavior of FFs. We slightly adapt this step in our implementation of the topological analysis because without this adjustment, our obfuscation worked too easily: Instead of removing the FSM candidate as a whole if one FF does not show any control behavior [Fyr+18], our implementation only removes the FF itself. Due to performance limitations of our topological analysis implementation, for the *picorv32*, the output control behavior calculation could not be finished for each sFF candidate. Thus, Table 5.5 shows the results for *picorv32* without performing the last step, the output control behavior calculation. However, the actual obfuscation results are assumed to improve further because this last step would remove additional FFs of FSM candidates.

For the FP approach, we apply one of the two FSM redesign methods on an arbitrary sFF of the original FSM: Redesign_Struct for the one-hot-encoded designs, and Redesign_Func for

the binary-encoded design. Compared with the evaluation in the "*RELIC-Tarjan*" paragraph, we insert the FSM-HP differently, which demonstrates a second FSM-HP insertion technique. We add the same FSM-HP to all designs; only the inputs are changed to match the inputs of the original design. The FSM-HP has five state bits and fulfills the features of the topological analysis as best as possible, including FPs, an SCC, and good influence/dependency and control behavior.

Applying topological analysis on these obfuscated designs leads to no or only partly identified sFFs of the original FSM (zero or low sensitivity), and instead of that, to successfully identified sFFs of the FSM-HP (perfect sensitivity), see columns three and four of Table 5.5. Consequently, the output of the topological analysis on these obfuscated designs can lead to one of the following two results:

- No FSM candidate contains any sFFs of the original design. As a result, data FFs or the FSM-HP FFs will be used to extract an incorrect FSM.
- An FSM candidate contains parts of the original sFFs. As a result, the subset of sFFs or data FFs or FSM-HP FFs will be used to extract an incorrect FSM.

Both cases successfully prevent the correct extraction of the original FSM and thus allow a successful obfuscation.

If an attacker is aware of the FP approach, he/she could try to post-process the final FSM candidates. The attacker would have to test every FSM candidate that emerges when adding every combination of FFs, which were not added to an FSM candidate by the topological analysis, to each of the FSM candidates. This exponentially increases the attack's complexity with each extra FF that could be added. In addition, after extraction, the attacker has to distinguish the correct FSM from all other FSM candidates, including FSM-HPs. However, this will not be feasible for plausible-looking FSM-HPs.

Overhead

This section discusses the cell area and delay overhead of the developed two-part FSM obfuscation, i.e. of adding an FSM-HP and of translating the original FSM to an unattractive FSM. We measure the cell area and the timing with proprietary EDA tools, assuming a frequency of 20 MHz.

On average, the obfuscated designs in the "*RELIC-Tarjan*" paragraph have 51%, the obfuscated designs in the "*Topological Analysis*" paragraph have 8% more cell area than the not obfuscated designs, see the results in Table 5.6. Thus, the generated overheads are larger and smaller than the measured overheads of a recently published FSM obfuscation method in [RB22] which reported 24% area overhead on average, or both smaller than another recently published FSM obfuscation method in [LO23] which reported 288% area overhead on average. Compared to the topological results, the RELIC-Tarjan results have a significantly larger overhead. We assume the following two reasons for this:

- The dissimilarity approach replicates the entire next state logic in the gate-level netlist when replicating bits, like state or counter bits, in the RTL code.
- For the "*RELIC-Tarjan*" experiment, we decided to add an FSM-HP by copying and modifying the original FSM of the design. Depending on the design, it was better or worse to isolate the FSM from the remaining design description before copying it. Thus, for some designs, like *siphash*, *sha1_core*, or *fpSqrt*, we selected one of only few existing design modules and copied it completely. Table 5.6 shows large area overheads for these

Table 5.6 Cell area and delay overhead results for the novel two-part FSM obfuscation

	Dissimilarity approach		FP approach	
Design	Area (%)	Delay (%)	Area (%)	Delay (%)
uart	-	-	27.72	-0.80
mem_int.	-	-	8.03	-1.09
siphash	77.71	0.08	-0.27	-0.68
sha1_core	100.07	-1.72	-	-
aes_core	3.71	0.08	-	-
altor32_l	5.72	-0.41	-	-
fpSqrt	99.00	-2.09	-	-
gcm_aes	19.34	0	-	-
picorv32	-	-	-4.35	-5.42
Average	**50.93**	**-0.68**	**7.78**	**-2.00**

designs. Choosing another FSM-HP insertion strategy might reduce the area overhead significantly.

As the obfuscation targets the FSM and an FSM is usually the smallest part of a design, our measured average overhead results should decrease for larger industrial designs. Overall, the obfuscation overhead, for example, depends on the number of replicated bits, on the number of FSMs in a design that must be obfuscated, on the size of the added FSM-HP, and on the size of the original FSM—which varies significantly less than design sizes. For designs with multiple FSMs, the designer can decide to only obfuscate security critical or proprietary FSMs to decrease the area overhead. In addition, when obfuscating multiple FSMs, the designer can choose between adding a single FSM-HP for all unattractive FSMs or adding an FSM-HP for each unattractive FSM. The first option results in a smaller area overhead.

In contrast to the area overhead and in contrast to recent obfuscation methods [RB22; LO23], the timing, i.e. the circuit slack, is not affected negatively by the novel two-part FSM obfuscation method. For our benchmarks, on average, the slack time is even decreased, resulting in 0.7% less slack time for the obfuscated designs in the "*RELIC-Tarjan*" paragraph and 2% less slack time for the obfuscated designs in the "*Topological Analysis*" paragraph, see the results in Table 5.6.

5.4 Summary

This chapter presents two novel approaches to obfuscate FSMs against RE.

The first approach, called FSM-TC, can prevent an extraction of the correct FSM with state-of-the-art sequential RE methods by applying Timing Camouflage on sFFs. Since the method relies on camouflaging, it does not require potentially attackable, secret locking keys. To enable Timing Camouflage on sFFs, we developed two FSM redesign methods. These redesign techniques might also be usable beyond FSM-TC, as shown in this section's second novel obfuscation approach. As a result, after obfuscation, the extracted FSMs have wrong transitions or states. The detailed analysis of the demonstration results showed that appropriately selected, targeted sFFs lead to a successful obfuscation.

The second approach presents a two-part state machine obfuscation technique to prevent sequential RE: hardware FSM honeypots and unattractive FSMs. This method does neither rely on potentially attackable, secret locking keys nor on foundry-enabled camouflaging techniques. Using a similarity-based and a topological-analysis-based sFF identification method, we demonstrate that state-of-the-art RE tools favor the more attractive FSM-HPs or cannot correctly identify the unattractive original FSMs. This leads to a successful obfuscation of the original FSMs and allows control over what will be identified by the attacker. The obfuscation approach is extendable by investigating other RE tool assumptions and features that can be exploited for unattractive FSMs. Also, the obfuscation can be increased if more than one honeypot is added to a design. In addition, if new identification mechanisms with new features are developed, the FSM-HP generation can be adapted accordingly, and new techniques for translating the original FSM into an unattractive FSM can be investigated. Thus, the obfuscation approach has the potential to be secure also for novel identification techniques. Additionally, the dissimilarity approach offers the possibility of being used as a fault detection technique. If a fault flips a bit of an FSM state that has replicated state bits, one can detect this fault by verifying if all replicated state bits have the same value. This would allow the dissimilarity approach to act as two protection mechanisms simultaneously: an obfuscation technique and a fault detection technique for FSMs.

Overall, FSM-TC once more demonstrates that combining different approaches of obfuscation can result in advantageous, new solutions. On the other hand, FSM-HP shows once again that a detailed investigation of the attack target, in this case, the FSM features, and of the attack tools, in this case, the RE tools, can also result in advantageous, new solutions.

6 Logic Locking Induced Fault Attacks

This chapter first presents the logic locking induced fault attack technique, including its motivation and attacker model. Next, it analyzes its applicability in detail by investigating the output corruption behavior of logic locking techniques, key management methods, and fault analysis techniques. Finally, a logic locking induced fault attack is demonstrated on an AES implementation using an extended Persistent Fault Analysis (PFA).

Parts of this chapter have already been pre-published in the author's works [Bru+20]. However, in particular, the applicability analysis in Section 6.2.1 was extended by a classification of locking approaches and by the analysis of further logic locking methods, while the use case in Section 6.3 was extended by a workflow schematic of the two-phase attack scenario. All authors of the work in [Bru+20] contributed to the discussions of the paper's idea. In particular, Michael Gruber and the author of the dissertation jointly elaborated the use case in Section 6.3. The author performed the logic locking with an insertion tool developed by Paul Thiele in his master thesis [Thi20], measured the faulty outputs with a Tcl script based on a Tcl script from Michael Mildner's master thesis [Mil19], and analyzed the success probability, while Michael Gruber developed and performed the two-phase PFA attack.

6.1 Introduction, Motivation, and Attacker Model

Logic locking is a protection technique to secure circuits against various supply chain attacks, like RE, overproduction, or IP theft [Kam+22]. Logic locking adds locking gates and key inputs into the design netlist. If the correct locking key is applied, the design works as intended. If a wrong locking key is applied, the design's functionality is corrupted. If a single wrong locking key bit is applied, one or multiple wires of the original design can be corrupted. This changed wire signal may propagate to one or more outputs, which exactly translates to a fault insertion.

This work is not meant as another extension to the number of possible attacks on logic locking methods. It does not aim for the recovery of the locking key but uses locking as a fault attack enabler and as integrated HT. Therefore, it is not comparable to other logic locking attack methods. The scheme changes the perception from a securing logic locking method to a possible security vulnerability. This raises the problems of logic locking to a new dimension because it does not only run the risk of being broken but can even be used as attack instrumentation.

Parallel to this work, the authors in [Hu+19] came up with the same technique but with a different attacker model and scope. They assume a malicious designer who knows the design implementation can change the original design and can insert locking gates at specific locations. This enables powerful attacks, like the insertion of HTs which are triggered by specific wrong locking key bits, the insertion of malicious FSM states which are reached by wrong locking input pattern, or the injection of faults by wrong locking key bits to extract cryptographic keys, bypass FSM transitions, or trigger HTs.

In the following, it has to be distinguished between the secrets of various research areas. The *locking key* K belongs to the hardware obfuscation field, whereas the secret information targeted by a fault attack is denoted by the cryptographic key κ.

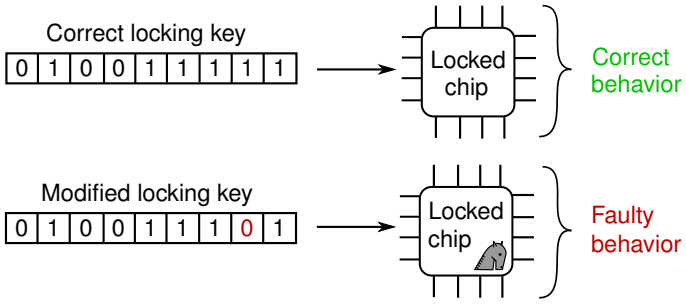


Figure 6.1 Logic locking induced fault or enabled HT ©2020 IEEE [Bru+20]

Attacker model: Adding an additional wire to a circuit that is controllable from the outside, e.g. a locking key input, opens the door to easily inserting faults or HTs. Suppose an attacker is able to insert a modified locking key. This can result in a fault at the corresponding key gates, i.e. a logic locking induced fault or a logic locking enabled HT, shown in Figure 6.1. Broken, loosely, or wrongly implemented logic locking and key management can lead to an attacker being able to insert such a modified locking key in the first place. Different from [Hu+19], we choose a more restrictive attacker model: The attacker does not have to be an untrusted designer who is able to take action on the design itself, but the attacker can use the unchanged, existing design. Consequently, the attacker model, which is assumed for the following analysis, is defined as:

Definition 15 (Logic Locking Induced Fault Attacker Model) *The attacker is able to modify at least each key bit once and either reset it or replace the device with a new one and start over. The key bit modification is done before the actual computations start. Additionally, the attacker is able to apply user-defined external inputs and observe the corresponding outputs.*

There are two possible scenarios for how to achieve the above-described attacker model:

1. The first scenario assumes an unknown locking key for the attacker. Even so, he can apply the attacker model in Definition 15 if a locking key bit can be flipped. This is the case if the key management enables an undetectable modification of the locking key before or while it is stored on the system or while it is provided to the system. A key bit flip is also achieved if the attacker identifies and inverts one of the locking key input wires. Only if the key is stored in a secure, not read- and write-able memory by the product owner itself, and the locking key input wires are not distinguishable from other wires, changing it becomes impossible.

2. For the second scenario, we assume that the locking key is known to the attacker, either due to successful attacks [Kam+22], e.g. powerful SAT-based [SRM15] or Satisfiability Modulo Theories (SMT) attacks [Aza+18], mask modification [EHP19], or probing [Rah+20a], or due to leakage. This breaks the design obfuscation but has no effect on the processed data. However, if an attacker inserts a modified locking key instead of the correct one, this induced fault might now exploit secret, processed data, like an AES key. Suppose the same locking key is used for more than one end product. In that case, the following situation becomes critical: the manufacturer overproduces chips, and locked chips without keys are on the market. If criminals get multiple such chips and the correct locking key, the attacker model in Definition 15 can again be applied by storing the differently modified keys in the one-time programmable memory of the individual chips. Finally, they can sell chips with a modified locking key and compromise systems using

such chips. In contrast, unique chip keys are usually used to prevent overproduction because this enables the IP or IC owners to control the number of activated products. If the unique locking key is used, its confidentiality might not be a target after the chip is selected for activation. Without a secure testing infrastructure, third-party factories might increase the erroneous testing outcome in order to use the remaining products on their own. For these products, a modified locking key can then be inserted.

The two scenarios show that a proper locking mechanism and key management can prevent the applicability of the attacker model in Definition 15. We also add a more detailed discussion about different approaches and their effect on the applicability of the new attack in Section 6.2.

Further applications: Recently published works show further scenarios on how to realize the concept of logic locking induced fault attacks or further attacker models of the concept of logic locking induced fault attacks. This also demonstrates its severity and its wide range of applicability.

Xu et al. [Xu+23] apply the concept of logic locking induced fault attacks on Neural Networks (NNs) to insert an HT. The authors assume the attacker to be the foundry. The malicious foundry figures out a suitable wrong locking key which allows a wrong classification when specific inputs are applied to the NN. Thus, the foundry generates an NN with a backdoor, which is activated by these specific inputs.

Upadhyaya et al. [UGP23] demonstrate the concept of logic locking induced fault attacks on cryptographic circuits. The authors assume that the end user has access to the correct locking key and can modify it, while he/she has no access to the correct cryptographic key. They present two approaches: The first approach is similar to our demonstration but uses differential cryptoanalysis instead of PFA [Zha+18a]. The second approach injects conventional faults in addition to logic locking induced faults, improving the success of key extraction.

6.2 Applicability of Logic Locking Induced Faults

Successful applicability of logic locking induced fault attacks mainly depends on the used locking methodology, the used key management or activation process, and the underlying fault attack analysis.

6.2.1 Exploring Characteristics of Output Corruption for Locking Methods

While Section 5.1 summarizes FSM obfuscation techniques and their threat models, this section explores the output corruption behavior of locking techniques. The characteristic of locking techniques is the usage of a locking key, which locks and unlocks the circuit [Kam+22]. Locking techniques (highlighted in dark blue) can be classified into *combinational locking* (highlighted in light blue) and *sequential locking* (highlighted in purple), see Figure 6.2. Combinational logic locking inserts locking gates together with locking key bits into the combinational part of the circuit [e.g. RKM08a]. In contrast, sequential logic locking concentrates on the sequential elements of a netlist, which means that the obfuscation concentrates on the control logic of a design, i.e. the FSM, [e.g. CB09], or on one or multiple FF(s) in the netlist [e.g. KCK20]. *Behavioral locking* (highlighted in yellow) contains combinational and sequential locking methods which also factor design behavioral characteristics into the key-based locking process [e.g XS17]. Also, if an *FSM obfuscation method* (highlighted in red) uses a secret locking bit sequence, it is classified as a sequential or behavioral locking technique. Consequently, not all FSM obfuscation methods are classified as locking techniques. Another subgroup of

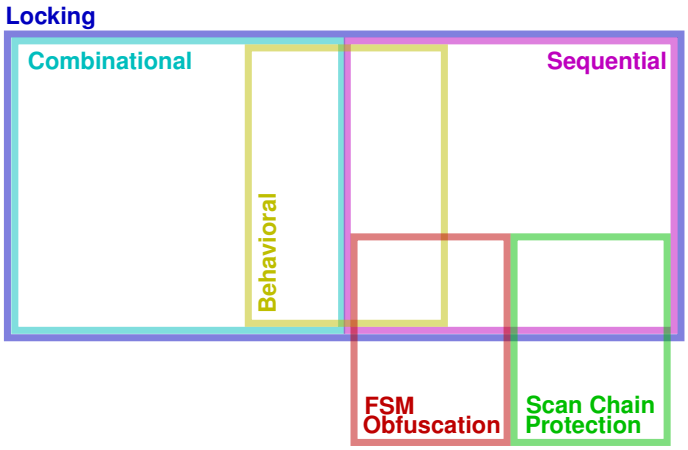


Figure 6.2 Classification of locking techniques

methods that receives special attention in literature is *scan path protection* (highlighted in green). Originally, scan path protection was developed to pretend side-channel attacks on crypto algorithms [e.g. PCB07]. Many scan path protection mechanisms corrupt the data only in the test mode but not in the functional mode [e.g. PCB07] or block the scan chain whenever the correct circuit functionality is executed [e.g. GZS18]. If an approach does not corrupt the original circuit, it might not be applicable to hinder RE or to allow exploitable logic locking induced fault attacks. If an approach can also protect the circuit itself and additionally uses locking key bits [e.g. KCK20], it is classified as a sequential locking technique. In addition, one could define many further subsets of locking methods, which is, however, not the focus of this work.

The following section analyzes the applicability of logic locking induced faults on different types of locking methods. It predominantly concentrates on a selection of early combinational logic locking methods to show essential concepts of the applicability of logic locking induced fault attacks. This does not include a discussion about the security of the presented locking methods. The crucial evaluation criterion for the applicability of logic locking induced faults is the degree of output corruption whenever one key bit is flipped. Regarding this evaluation criterion, three main categories of output corruption behavior are defined as follows:

- *Flip:* A key bit flip results in a flipped output signal.
- *Constant:* A key bit flip results in a fixed output signal value, i.e. '0' or '1'.
- *Dependent:* A key bit flip results in an output corruption or in no output corruption, depending on further circuit-specific properties, like other key bits, the design's functionality represented by its circuit structure, or inputs.

A *flip* and a *constant* output corruption are highly applicable for logic locking induced fault attacks because both guarantee an injected fault whenever a key bit is flipped. A *dependent* output corruption is still—however less convenient—applicable for logic locking induced fault attacks because a flipped key bit does not guarantee that the output is corrupted and that a fault is introduced, see Section 6.2.3. Nevertheless, the presence or absence of introduced faults is deterministic with fixed circuit-specific properties, which depend on the applied locking method. Thus, to apply logic locking induced fault attacks, some of the circuit-specific properties are fixed while the external inputs are varied. Then, for each input pattern combination, a locking key bit is modified. In contrast to the *flip* or *constant* output corruption, the disadvantage of the

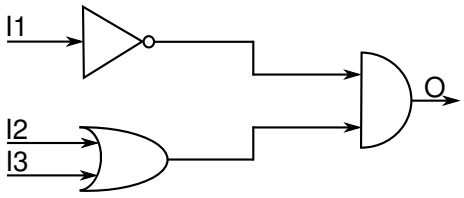


Figure 6.3 Original example netlist structure ©2020 IEEE [Bru+20]

dependent output corruption is that more runs are required and that a fault injection is linked with the selected input pattern.

To illustrate the different approaches of logic locking, where applicable, the example circuit in Figure 6.3, in particular its output signal O, is locked, resulting in a locked output signal O'. The example circuit's inputs are referred to as I={I1,I2,I3}.

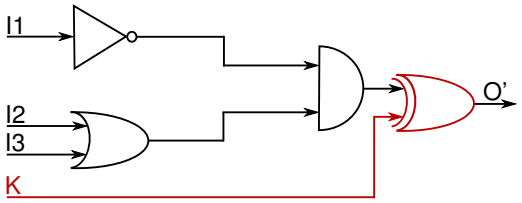
Combinational Logic Locking Methods

The first combinational logic locking technique introduces an XOR gate or XNOR gate at a random location in the netlist and defines the second input as key bit [RKM08a], see Figure 6.4a. As a result, the corruption of the output only depends on the applied key bit and is thus a *flip* output corruption. The same is true if a specific insertion rule is used [Yas+16b; Raj+12a; Raj+15; Raj+12b] or if the scan path is disabled after the key is inserted [Gui+16; GZS18; Lim+19]. Depending on the optimization goal, an insertion rule places the locking elements in the circuit, e.g. to achieve an output corruption of 50% hamming distance [Raj+15; Raj+12a], or to prevent additional attacks, like the sensitization attack [Raj+12b] or the logic cone based attack [LT15], [Raj+12b; Yas+16b; LT15]. As a result, fault insertion points are no longer randomly distributed over the circuit. Although the justified selection of insertion points has no effect on the output corruption behavior, they can affect the exploitability of logic locking induced faults.

Most of the remaining locking techniques show a much more complex output corruption behavior, thus having a *dependent* output corruption. The following analyzes examples of such more complex locking techniques in more detail.

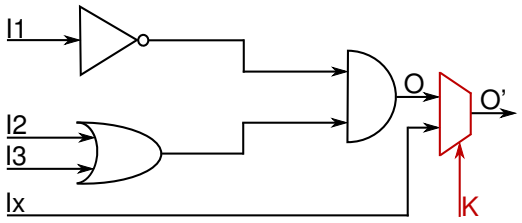
Some approaches use multiplexers instead of XOR/XNOR gates, like in [Raj+15], [PM15] and [GRR12], or suggest a combination of both gate types [LT15]. In [Raj+15] and [PM15], the XOR/XNOR gate is mainly substituted by a multiplexer, see Figure 6.4b. The method in [GRR12] duplicates the circuit, modifies the duplicate's input, and adds a key-controlled multiplexer to select the output of the unmodified circuit when the correct key is applied, see Figure 6.4c. To modify the duplicated circuit, one of its inputs is corrupted using a switching circuitry controlled by process variation sensors, exemplified by inverters, a multiplexer, and a PUF in Figure 6.4c. Thus, as explained above, the output corruption of both techniques does not solely depend on the applied locking key bit. Instead, for these methods, the output corruption also depends on the provided input pattern and the respective logic cone. Consequently, the circuit, including the PUF response, has to remain unchanged while a key bit is modified for each possible input pattern.

Instead of using multiplexers as alternative to XOR/XNOR gates, also Lookup-Tables (LUTs) can be used, see for example the work in [BTZ10], [Kam+19], [Mar+18], [Kol+19], or [Kam+20]. A LUT is a two-column table that provides one or multiple predefined bit values (output values) for one or multiple applied bit values (entry values). In contrast to many other locking methods,



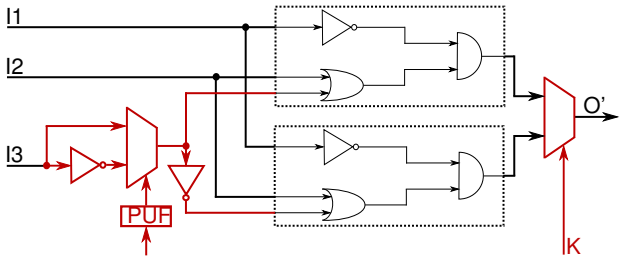
K	O'
$k_✓$	✓
$k_✗$	✗

(a) Using an XOR gate with key input bit K at random [RKM08a] or specific [Yas+16b; Raj+12a; Raj+15; Raj+12b] locations ©2020 IEEE [Bru+20]



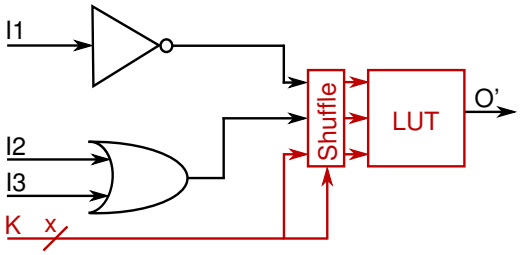
K	Ix	O'
$k_✓$	Ix	✓
$k_✗$	O	✓
	Ix\{O}	✗

(b) Using a multiplexer with key selector bit K [Raj+15], [PM15]



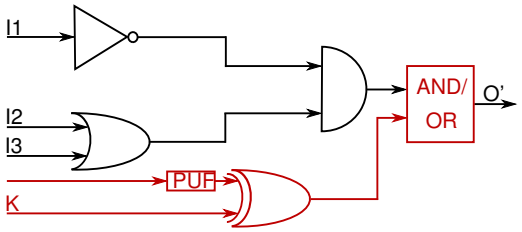
K	O'
$k_✓$	✓
$k_✗$	dep. on I & circuit

(c) Using a duplicated circuitry controlled by a switching circuitry, and a multiplexer controlled by a key input bit K [GRR12]



K	O'
$k_✓$	✓
$k_✗$	dep. on I, K, circuit, Shuffle and LUT

(d) Using a LUT and shuffling with additional inputs and shuffle key K (provided by a PUF) [Mar+18]



K	O'
$k_✓$	✓
$k_✗$	0/1

(e) Using an AND or OR gate together with a PUF and key input bit K [Dup+14] ©2020 IEEE [Bru+20]

Figure 6.4 Typical combinational logic locking methods: example netlist structures, and the corresponding evaluation tables by applying the correct key bit $k_✓$ and the incorrect key bit $k_✗$

LUTs are usually not added as extra components to the netlists but replace one or multiple existing gates or wires. While in some works [e.g. BTZ10; Kam+19] the LUTs are a one-to-one representation of the replaced gates, in other works [e.g. Mar+18; Kol+19] the LUTs have more inputs than the replaced gate, but can also preserve the original functionality, see Figure 6.4d. To assign the additional inputs, one can use wires of the original circuit, dummy wires, or new inputs. LUT-based locking techniques often use wire re-routing or shuffling as additional obfuscation, e.g. in [Kam+19], [Mar+18], and [Kam+20], while wire re-routing and shuffling can also be applied as an obfuscation technique on its own, see for example the work in [Sha+18].

Regarding the key-controlled re-routing or shuffling techniques, the following output corruption can be observed. For a wrong key bit in [Kam+19], two input wires are re-routed or one input wire is inverted. If a key bit responsible for the second effect is flipped, this results in a *flip* output corruption, like for XOR/XNOR locking. For a wrong key bit in [Kam+20], two input wires are re-routed or one bit is replaced by the output of a Boolean function. For this case, if a key bit responsible for the second effect is flipped, the output corruption depends on the provided input pattern and the used Boolean function. For a wrong key bit in [Mar+18], the output corruption depends on the other key bits (provided by a PUF), the provided input pattern, the logic cone, the shuffle instance, and the LUT. For this *dependent* output corruption behavior, again, all additional variables have to remain unchanged while a key bit is modified for each possible input pattern.

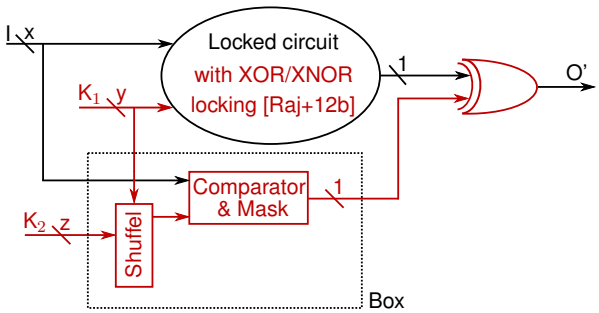
Regarding the LUT techniques themselves, the following output corruption can be observed. The configuration bits of the LUT, i.e. the pairs of entry and output values, can be interpreted as locking key bits and can thus be modified instead of the locking key bits. If an output value is flipped, the output corruption behavior and thus the applicability of logic locking induced fault attacks is comparable with XOR/XNOR locking methods. In contrast, if an entry value is flipped, the output corruption behavior is less controllable from a fault attack point of view.

Several approaches further developed the idea of using LUTs [Col+22]. For example, the authors in [Moh+21] propose to use embedded FPGAs instead of LUTs. An FPGA is composed of a routing structure and Complex Logic Blocks (CLBs) which themselves are composed among others of LUTs.

Another method [Dup+14] suggests AND/OR gates instead of XOR/XNOR gates to extend logic locking with HT prevention, see Figure 6.4e. Using AND/OR gates minimizes the number of attractive insertion places for HTs. In addition, the method modifies the locking key input of the AND/OR gates using an XOR gate and a PUF. This enables device-dependent locking keys. Flipping a locking key bit for an unchanged PUF response results in a constant value for the output gate signal O' instead of a flipped value. For an AND gate, a key bit flip results in a constant '0', while for an OR gate, a key bit flip results in a constant '1'. Thus, this method shows a *constant* output corruption.

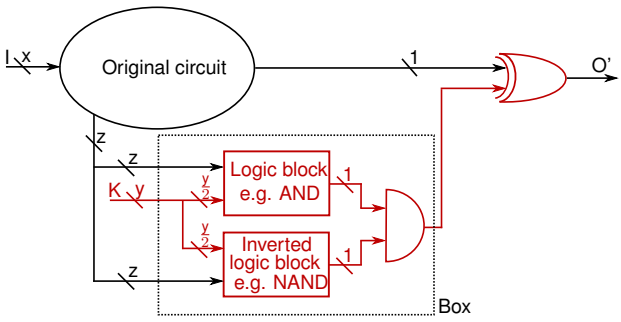
In 2015, the authors in [SRM15] presented a new attack method, the SAT-based attack. This caused the development of another group of locking methods: methods to withstand this new attack type. SAT-based attacks enable the fast elimination of impossible key sets by exploring input patterns. As a result, a race to develop new, hardened logic locking methods and improved SAT-based attacks began.

The first new, hardened logic locking methods add additional locking blocks together with an XOR gate to increase the complexity of SAT-based attack [Yas+16a; XS16]. The locking blocks affect the output corruption behavior, such that an incorrect key does not necessarily introduce a fault at the affected wire. Instead, the wire will only be faulted for one or a small number of input value combinations. In [Yas+16a], the logic locking scheme of [Raj+12b], using



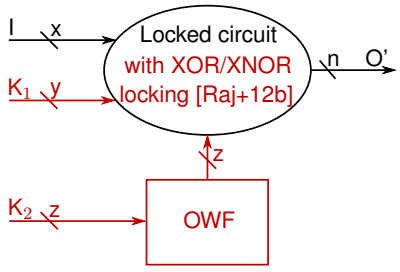
K_1	K_2	I	Box output	O'
$k_{1\checkmark}$	$k_{2\checkmark}$	I	0	✓
	$k_{2\times}$	Shuf.$(k_{2\times} \mid k_{1\checkmark})$	1	✗
		I\{Shuf.$(k_{2\times} \mid k_{1\checkmark})$}	0	✓
$k_{1\times}$	$k_{2\checkmark/\times}$	Shuf.$(k_{2\checkmark/\times} \mid k_{1\times})$	1	dep. on K_1 & locked circuit
		I\{Shuf.$(k_{2\checkmark/\times} \mid k_{1\times})$}	0	dep. on K_1 & locked circuit

(a) Using a SARLock box with input keys K_1 and K_2 [Yas+16a]



K	Box output	O'
$k_{\checkmark}$	0	✓
$k_{\times}$	dep. on I, K, orig. circuit & logic block	

(b) Using a type-0 Anti-SAT box with input key K [XS16]



	K_2	O'
No K_1*	$k_{2\checkmark}$	✓
	$k_{2\times}$	dep. on OWF & locked circuit

* The usage of K_1 is shown in Figure 6.4a.

(c) Using an OWF and input keys K_1 and K_2 [Yas+16b]

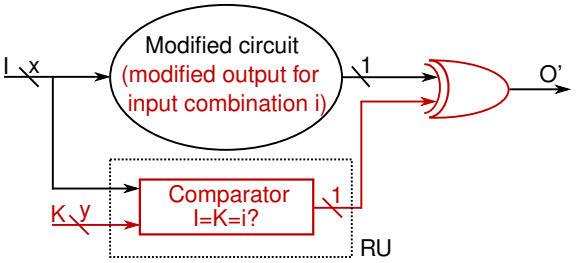
Figure 6.5 Combinational logic locking methods to withstand the SAT attack: example netlist structures, and the corresponding evaluation tables by applying the correct keys, $k_{\checkmark}$, $k_{1\checkmark}$, $k_{2\checkmark}$, and incorrect keys, $k_{\times}$, $k_{1\times}$, $k_{2\times}$

the locking key K_1, is combined with a new locking block, called SARLock, using the locking key K_2, see Figure 6.5a. If one bit of the locking key K_2 is flipped, the SARLock block outputs the value '0' except for one specific input combination. If the correct locking key K_2 is applied, the output of the SARLock block is always '0'. To achieve the above-described behavior, the SARLock block is composed of a comparator and a mask. The output of the SARLock block is xored with the original circuit output. To combine the SARLock block with the previous locking method of [Raj+12b], K_2 is shuffled with K_1 before it is applied to the comparator and mask. The specific shuffling technique is not prescribed by the authors. For instance, they suggest to permute the bits of K_2 based on K_1, as in [RKM08b]. Similarly, in [XS16], a locking block, called Anti-SAT, is added, see Figure 6.5b. The type-0 Anti-SAT block outputs the value '0' if the correct locking key K is applied. For a wrong locking key, the output varies based on the input combinations. Finally, the output signal is xored with the original circuit's output. In contrast, the type-1 Anti-SAT block outputs the value '1' for the correct locking key and '0' or '1' for a wrong locking key. In addition, the authors in [XS16] suggest adding other logic locking methods to the original circuit or to the Anti-SAT block. Again, these methods show a *dependent* output corruption. The additional key bits, the circuit structure, the input pattern, the shuffle instance, and the logic blocks have to remain unchanged while a key bit is modified for each possible input pattern.

Other post-SAT locking methods explicitly exploit the limitations of SAT-based attack. In [Yas+16b], the work of [Raj+12b] is extended by adding an One-Way Function (OWF) in front of the locked netlist to generate non-retraceable outputs, see Figure 6.5c. Here, the output corruption depends on the OWF, multiple locking key bits, the input pattern, and the locked circuit structure. However, a key bit flip of K_2 leads to an uncontrollable number of bit flips of the OWF output. This makes the applicability of logic locking induced fault attacks more difficult.

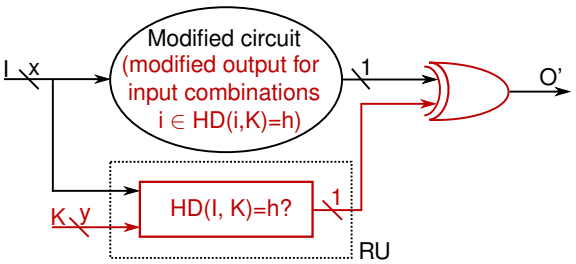
While the SAT attacks improve and additional attacks, including signal probability skewing [Yas+17a], bypassing [Xu+17], or removal attacks [Yas+20], are applied, the logic locking methods strengthened even more. One important development was to prevent the existence of an unchanged, original circuit while solely adding extra locking blocks for the locking technique.

Different techniques have been developed that modify the original circuit for a single input pattern [Yas+17c], [Yas+17b], for multiple input patterns [Yas+17b], or for an arbitrary number of input patterns [CS22]. They all use a restore unit and an XOR gate to reconstruct the modified circuit output. The reconstruction is successful if the correct locking key is applied. The restore units are composed of a comparator [Yas+17c], a hamming distance checker [Yas+17b], or a LUT [Yas+17b; CS22]. Figure 6.6a shows the concept of [Yas+17c]. The comparator outputs the value '1' if the input bits are equal to the locking key bits and '0' otherwise. The same input bits lead to a modified circuit output. Finally, the output of the comparator is xored with the original output. This restores the modified circuit output if the correct locking key is applied. The concept of [Yas+17b] is similar to the concept of [Yas+17c]. Instead of a comparator, it uses a hamming distance checker or a LUT as a restore unit. Using the hamming distance checker as a restore unit, the method restores input combinations that have a hamming distance of exact h to the locking key, see Figure 6.6b. This means that if a modified locking key is applied, the output O' is only corrupted for specific input patterns: input patterns that do not have both, a hamming distance h to the modified and to the correct locking key; and input patterns that do not have both, no hamming distance h to the modified and to the correct locking key. Using the LUT as a restore unit, the method restores specific input combinations, which are defined by the LUT, see Figure 6.6c. By applying the correct locking key, all the previously modified outputs for the corresponding input combinations are restored. In [ZSR19], the strategy of [Yas+17b] is strengthened by integrating an OWF, similar to [Yas+16b]. As input for the OWF either the



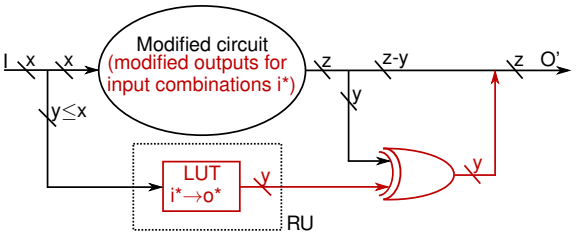
K	I	RU	O'
$k_✓$	$k_✓$	1	✓
	$I \setminus \{k_✓\}$	0	✓
$k_✗$	$k_✗$	1	✗
	$k_✓$	0	✗
	$I \setminus \{k_✓, k_✗\}$	0	✓

(a) Using a comparator with key K as restore unit [Yas+17c]



K	I	RU	O'
$k_✓$	$HD(I,k_✓) = h$	1	✓
	$HD(I,k_✓) \neq h$	0	✓
$k_✗$	$HD(I,k_✗) = h$ & $HD(I, k_✓) \neq h$	1	✗
	$HD(I,k_✗) = h$ & $HD(I, k_✓) = h$	1	✓
	$HD(I,k_✗) \neq h$ & $HD(I, k_✓) = h$	0	✗
	$HD(I,k_✗) \neq h$ & $HD(I, k_✓) \neq h$	0	✓

(b) Using a hamming distance checker HD with key K as restore unit [Yas+17b] ©2020 IEEE [Bru+20]



LUT	I	RU	O'
$LUT_✓$	i*	o*	✓
	$I \setminus \{i^*\}$	0	✓
$LUT_✗$	I	dep. on I and LUT	

(c) Using a LUT as restore unit [Yas+17b]

Figure 6.6 Combinational logic locking methods to withstand advanced SAT attacks and other attacks: example netlist structures, and the corresponding evaluation tables by applying the correct key $k_✓$ or LUT $LUT_✓$, and incorrect key $k_✗$ or LUT $LUT_✗$

key or the input signal is used. Similar to the majority of SAT hardened locking techniques, the approaches in Figures 6.6a and 6.6b have a *dependent* output corruption behavior and thus the applicability of logic locking induced fault attacks requires the following strategy: flipping a key bit for all possible input pattern combinations by keeping all other variables unchanged. Adding an OWF influences the applicability of logic locking induced fault attacks similarly to the method in [Yas+16b]. Using a LUT instead of a comparator or hamming distance checker shows a similar applicability than the previously introduced locking methods with LUTs [e.g. BTZ10; Mar+18].

Further Logic Locking Methods

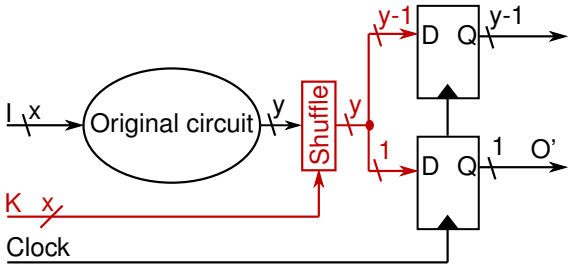
So far, only combinational logic locking methods—without behavioral locking techniques—were considered, see Figure 6.2. In the following, the remaining locking techniques are briefly discussed.

Many *FSM obfuscation* methods—including behavioral locking techniques—show special applicability of logic locking induced fault attacks. For most FSM obfuscation methods, applying a wrong locking key results in a transition to a wrong state [e.g. JS21a], what makes logic locking induced fault attacks basically applicable. However, many of these approaches ensure that wrong locking keys result in non-critical states or black hole states [e.g. DY18], making logic locking induced faults possible but not exploitable. Other FSM obfuscation methods (additionally) corrupt the FSM output or an arbitrary circuit output when a wrong key is applied [e.g. KTV19]. The output corruption behavior and, thus, the applicability of logic locking induced fault attacks of these methods is similar to the ones of the combinational locking methods.

All remaining *sequential locking* methods—without behavioral locking—modify the FFs' inputs [e.g. Mar+20] or modify the FFs' outputs [e.g. KCK20]. As a result, similar to combinational locking methods, also sequential locking methods show different output corruption behavior and thus applicability to logic locking induced fault attacks. As an example, the method in [Mar+20] shows a *dependent* output behavior, see Figure 6.7a. It adds a SAT-hardened permutation network or LUT in front of FFs and is thus comparable to the applicability of LUT-based locking methods, see Figure 6.4d. As another example, the method in [KCK20] shows a *flip* output behavior, see Figure 6.7b. It adds a key-controlled multiplexer to the inverted and not inverted FF output and is thus comparable to the applicability of the work in [RKM08a] in Figure 6.4a.

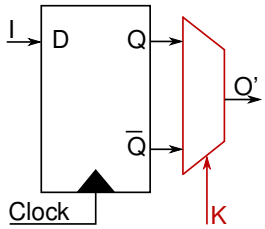
Finally, *behavioral locking techniques*, including combinational and sequential behavioral locking techniques, are analyzed. Some combinational behavioral locking techniques insert key-controlled dummy cycles into the circuit to explicitly exploit SAT-based attack limitations [e.g. RMS18; Sha+17]. The work in [Sha+17] introduces this idea and adds non-reducible cycles using key-controlled gates or multiplexers, see Figure 6.8a. Applying the correct key disables the inserted loop and maintains the correct functionality. Applying an incorrect key bit flips the locking gate or multiplexer output wire, depending on the circuit and input pattern. Thus, the method shows a *dependent* output corruption.

Other behavioral locking methods use circuit time properties for their extended locking mechanisms [e.g. XS17; Swe+20]. The combinational behavioral locking method in [XS17] extends the classical XOR/XNOR locking [RKM08a], which uses the locking key K_1, by a tunable delay buffer, see Figure 6.8b. A second locking key K_2 controls the tunable delay buffer, resulting in a correct or wrong path delay. Thus, by modifying a key bit of K_1, the applicability and output corruption is the same as for the XOR/XNOR locking method [RKM08a]. Instead, by modifying a key bit of K_2, the method shows a *dependent* output corruption, which depends on the inputs and the original and modified circuit path delay. The sequential behavioral locking method in [Swe+20] locks FFs by replacing them with two latches with key-controlled clock inputs,



K	O'
$k_✓$	✓
$k_✗$	dep. on I, K, Shuffle, circuit

(a) Using a shuffle network in front of FFs with shuffle key K [Mar+20]



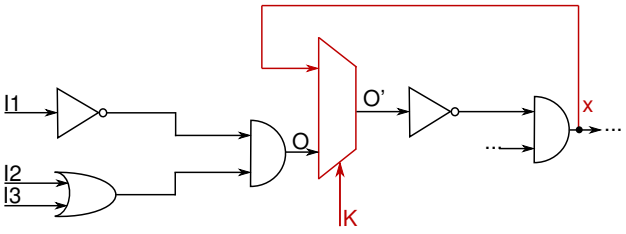
K	O'
$k_✓$	✓
$k_✗$	✗

(b) Using a multiplexer behind FFs with key selector bit K [KCK20]

Figure 6.7 Sequential logic locking methods (targeting sequential elements in general): example netlist structures, and the corresponding evaluation tables by applying the correct key $k_✓$ and incorrect key $k_✗$

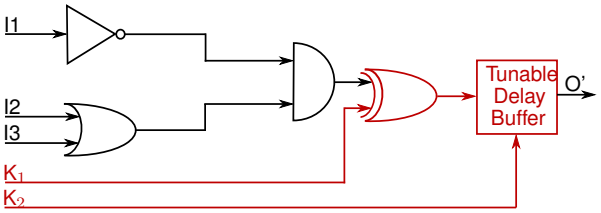
see Figure 6.8c. Depending on the selected key bits, the two latches act as a positive-edge clocked FF, negative-edge clocked FF, combinational path with extra delay, or combinational path without extra delay. The method shows a *dependent* output corruption. When flipping one key bit, the output corruption depends on the other key bit, the inputs, and the original and modified circuit path delay.

A final aspect is the combination of different locking methods, which is often favored and recommended [Yas+16a; Yas+16b]. In general, two strategies must be distinguished: a dependent and an independent combination. For an independent combination, the used locking keys can be used independently, while for a dependent combination, the locking keys influence each other. Integrating multiple locking methods intends to strengthen the circuit security, but this takes a turn for the worse by considering the presented logic locking induced fault attack. The approaches in [Yas+16b] or [XS17], for example, provide two independent keys where one drives the XOR/XNOR gates, whereas the other is fed through an OWF or a tunable delay buffer. While the XOR/XNOR locking technique provides good applicability of logic locking induced attacks due to its *flip* output characteristic, the added locking methods show a much more complex applicability and a *dependent* output characteristic. However, due to the independent usage of methods and, thus, locking keys, logic locking induced attacks can focus on each locking method and key independently. If the attacker modifies one bit of the XOR/XNOR locking key, he/she can make use of the *flip* output corruption, making logic locking induced attacks highly applicable. This shows that as soon as more than one logic locking strategy with an individual secret key is implemented, the more suitable one can be chosen for a fault attack.



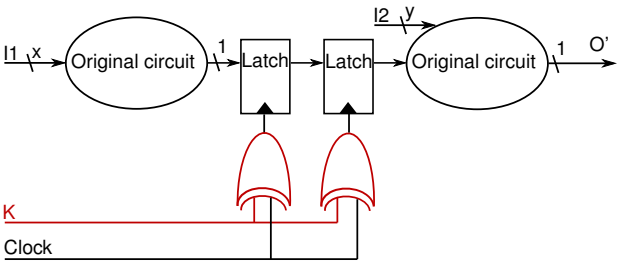
K	x	O'
$k_{\checkmark}$	x	✓
$k_{\times}$	O	✓
	x\{O}	✗

(a) Using a dummy cycle enabled by a multiplexer or a gate controlled by key K [Sha+17]



K_1	K_2	O'
$k_{1\checkmark}$	$k_{2\checkmark}$	✓
$k_{1\times}$	$k_{2\checkmark}$	✗
$k_{1\checkmark/\times}$	$k_{2\times}$	dep. on K_1, I, circuit path delay

(b) Using an XOR/XNOR gate controlled by key K_1 and a tunable delay buffer controlled by key K_2 [XS17]



K	O'
$k_{\checkmark}$	✓
$k_{\times}$	dep. on I1, I2, K, circuit

(c) Using two latches with locked clock input to implement positive-clocked FF, negative-clocked FF, or short or long delay [Swe+20]

Figure 6.8 Behavioral logic locking methods: example netlist structures, and the corresponding evaluation tables by applying the correct keys, $k_{\checkmark}$, $k_{1\checkmark}$, $k_{2\checkmark}$, and incorrect keys, $k_{\times}$, $k_{1\times}$, $k_{2\times}$

6.2.2 Exploring Characteristics of Key Management Methods

The previous section discusses the effect of different locking methods on the applicability of logic locking induced fault attacks. This analysis was done independently of how the modified locking key K will be introduced. The following section first states the difference to the storage of cryptographic keys κ. Then, it discusses the different approaches of locking or activation key management and storage and their effects on the applicability of logic locking induced fault attacks.

The challenges and possibilities of secret key storage are comparable between locking methods and crypto algorithms. However, the application and use case differ. Cryptographic keys κ are mainly inserted by end users themselves. Locking keys K are mostly designed to be placed by the chip owner, designer, or foundry and not by the end user. Consequently, the end user has to trust third parties and has no chance to detect the communication of a wrong locking key. Additionally, cryptographic keys can be changed frequently in order to minimize potential attacks. This protection mechanism is usually not applicable for locking keys because their bit

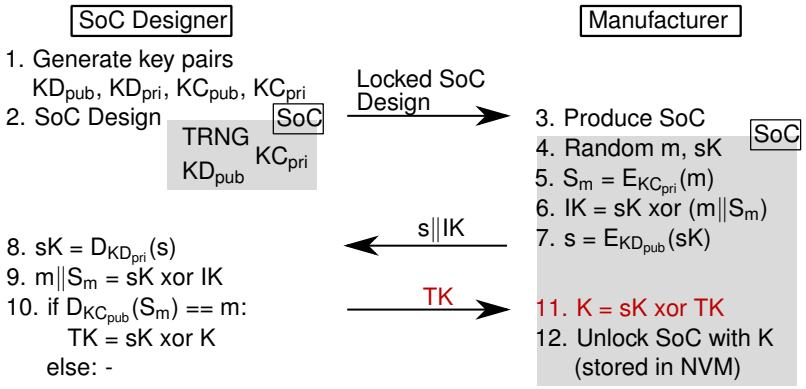


Figure 6.9 Activation protocol to prevent IC overproduction [Gui+16] does not hold integrity of locking key K ©2020 IEEE [Bru+20]

values are fixed by the manufactured hardware. Also, while for crypto algorithms the key is usually required once in the beginning, the locking key has to be valid for the complete runtime [EHP19]. This drastically increases the risk of a successful attack. Finally, the requirement to exploit the secret key vulnerability and, thus, the requirement for the key storage differs. For cryptographic and locking keys, if an attacker can assign an arbitrary, predefined value to a key bit, he/she can recover the actual, secret key bit's value. Instead, for locking keys, flipping a key bit is sufficient to make the method vulnerable to logic locking induced fault attacks.

Concerning the management of locking keys, the crucial criterion to evaluate the applicability is the integrity and confidentiality of the applied locking key. Secure activation or key management should prevent typical threats like IC overproduction. Thus, their protocols are often intended to ensure the confidentiality of the locking key, not the integrity of the locking key.

The first activation protocols considered a subset of the possible threats, like logic locking the IP piracy, or active IC metering the IC overproduction [Gui+16]. Consequently, for such protocols, at least one threat that was not considered exists. This threat can come from the foundry, the assembly, the IC designer, or the end user. Besides, many protection methods were also exploited by attacks, like SAT [SRM15] or SMT attacks [Aza+18], mask modification [EHP19], or probing [Rah+20a].

However, more recently, design flows are developed which want to achieve full coverage of all possible supply chain security threats [e.g. Gui+16; GZS18; Lim+19; Aza+19]. The approach in [Gui+16] prevents IC overproduction, IP overuse, and IP piracy by a hybrid cryptosystem infrastructure, which enables secure authentication and communication, as well as secure testing. In [GZS18] or [Lim+19], the prevention of possible supply chain threats is achieved by allowing the activation only in a secure environment, like by the IP owners themselves. Secure testing is achieved with scan path protection mechanisms, which, for example, ensure that the correct secret key will only be used during functional mode and with a disabled scan chain [GZS18]. The approach in [Aza+19] does not store the locking key on the untrusted, locked chip and provides a dynamic activation key. All these new approaches aim for the prevention of known supply chain threats and thus, amongst others, the confidentiality of the locking key. The integrity of the key is not aimed, even though it is achieved for some of them. For [GZS18] and [Lim+19], the locking key does not have to be transmitted. Consequently, it cannot be modified before it is inserted. In [Aza+19], an authenticated encryption is used to transmit the locking key. Thus, a modification will be identified. The approach in [Gui+16] uses a one-time pad encryption instead of an authenticated encryption to transmit the locking

key. Here, a modification of the transferred activation key TK stays undetected, as shown with the red colored parts in Figure 6.9. This enables the insertion of a bitwise modified locking key K.

A second aspect of key management that affects the applicability of logic locking induced fault attacks is the different types of key storage [Aza+19]:

- Permanently on the device, e.g. tamper-proof memory [e.g. Raj+15]
- Indirect permanently on the device, e.g. PUF [e.g. Dup+14]
- Outside of the device and loaded as needed [e.g. Aza+19]

The first two types rely on a secret or a hardware component on the device, like a secure memory or a PUF implementation. These elements have to be known to the manufacturer of the device, what represents a security risk. If the manufacturer is the attacker, he/she can manipulate the key storage such that, for example, a locking key bit is flipped. This makes logic locking induced attacks applicable. Similarly, also activation processes often rely on secrets on the device, like private keys, see the example in Figure 6.9. Depending on the activation process, manipulating these secrets might also ease the applicability of logic locking induced fault attacks.

The third type does not require permanent storage of the locking key on the device itself. This allows everyone who can provide a locking key to the device to modify the locking key. Depending on the activation process and its achieved integrity and confidentiality of the locking key, the third type can enable an easy applicability of logic locking induced fault attacks.

6.2.3 Exploring Characteristics of Fault Attacks

The last parameter to evaluate the applicability of logic locking induced fault attacks is the identification of a suitable underlying fault analysis technique, as different types of fault analysis techniques assume different types of fault models. One can classify fault models into the following three categories [Zha+18a]:

- *Transient fault*: Fault that is introduced into the running system once at a certain point in time.
- *Persistent fault*: Fault that is introduced into the running system once at a certain period in time.
- *Permanent fault*: Fault that is introduced during the complete runtime of the system.

A fault that is injected by logic locking methods, i.e. by a corrupted locking key, usually lasts longer, which points to a persistent or permanent fault. Persistent faults are used by the PFA, introduced by Zhang et al. [Zha+18a]. A permanent fault is similar to the PFA but with an extended period of fault injection time. Another special group of fault models exists, the stuck-at-zero/one faults. Stuck-at-zero/one faults are used by the Collision Fault Analysis (CFA), introduced by Blömer and Krummel [BK06]. This type of fault keeps an intermediate at a fixed value and thus corresponds to the previously defined *constant* output corruption behavior. It is applicable for logic locking methods that corrupt the netlist wire to have a fixed value, i.e. zero or one, like is the case for the locking method in [Dup+14].

However, in general, the applicability of the fault analysis mainly depends on the following three properties:

1. Sufficient data has to be collected to enable the required statistics for the fault analysis model.

2. The location in the gate-level netlist where the locking gates are inserted has to be exploitable for the fault analysis model.

3. The output corruption behavior of the locking method has to allow properties 1. and 2.

If the inserted fault is predominantly masked, i.e. the injected fault is not distributed to the netlist outputs, too less corrupted data might be collected, which hinders the applicability of a successful fault analysis attack. Reasons for predominantly masked faults are a disadvantageous position of the locking gates in the gate-level netlist or a locking method with a *dependent* output corruption behavior. A *dependent* output corruption behavior corrupts outputs only for some specific input patterns or circuit-specific properties what corresponds to predominantly masked faults. Thus, locking methods with a *dependent* output corruption behavior are less applicable than locking methods with a *flip* output corruption behavior.

6.3 Use Case: Locked Crypto Core

The following use case shows the applicability of the new attack on a locked crypto core. A crypto core might be locked because it is part of a larger design or because its implementation is licensed. If the logic locking method enables a fault injection at a specific location of the implementation, an attacker can use this to extract the secret cryptographic key κ. Different from [Hu+19], the locking gates are not placed intentionally at specific locations, and the locking key does not have to be varied during runtime. Our attack will be carried out based on a permanently faulted AES S-box. To demonstrate this, a modified AES encryption implementation from [Uss02] was synthesized with QFLOW [Ope19], together with the 0.35μm OSU library [Okl15]. The used implementation does not expand the 128-bit key κ on the fly but inserts an already expanded master key μ which is composed of the required AES roundkeys. Extracting all correct AES roundkey bytes μ_j is equivalent to extracting all correct AES key bytes κ_j, as the key expansion can be easily reversed. In addition, the used AES design has 16 S-boxes (one for each byte), which are implemented as transition tables in case statements. The synthesis translates them together with the remaining implementation into combinational circuits. Finally, we randomly insert XOR/XNOR gates [RKM08a] to lock the circuit. To enable the applicability of the new attack for this implementation, the following assumptions must hold:

1. The key management of the locking key K enables key bit flips. The secret AES key κ is stored in a secure, unreadable memory.

2. The logic locking is the same for all produced chips, i.e. an equal logic locking gate insertion and an equal locking key.

3. For a template phase, the attacker is able to acquire a locked chip, including the crypto core, where he/she can choose the plaintext, ciphertext, and AES key by himself/herself.

4. For an attack phase, the attacker is able to acquire a locked chip, including the crypto core, from an external instance, e.g. a company, who securely stored their expanded, secret AES key on the core.

6.3.1 Probability of Locking Gate Insertion at Suitable Places in AES Core

First, the use case investigates if the probability of inserting a locking gate at a suitable place is high enough. Here, a suitable place is defined as a place in the AES core which makes the AES vulnerable to PFAs. We assume that random XOR/XNOR locking can insert a locking

gate only once on a unique wire and not on input and output wires. As an AES core can have more than one suitable place, the probability P_z^* is defined as follows.

Definition 16 (Probability P_z^*) *The probability P_z^* describes the probability that at least one suitable wire is locked when inserting z locking gates.*

To calculate the probability for x suitable placements over z inserted locking gates of a netlist with w wires (s suitable ones and $(w-s)$ not suitable ones) without replacement, the hypergeometric distribution $P(x)$ is used. Consequently, the probability P_z^* in Definition 16 is calculated with the aid of probability $P(x=0)$, i.e. no locking gate is placed at a suitable wire, subtracted from one.

$$P_z^* = 1 - P(x=0) = 1 - \frac{\binom{w-s}{z}}{\binom{w}{z}} \tag{6.1}$$

To identify which of the locking key bits drives a suitable wire, the attacker can iterate over all of them. The more suitable places are affected by locking gates, the fewer locking key bits have to be tested in order to identify a suitable one.

The synthesized AES netlist contains $w = 21474$ wires, excluding input and output wires. The number of locking gates that are inserted is chosen to be 10% of the overall number of nodes, including input and output nodes. This results in $z = 2327$. The number of suitable or interesting wires for placing a locking gate mainly depends on the applied fault attack method. For the following example, a fault attack based on an S-box manipulation is considered [Zha+18a]. For each manipulated S-box in the synthesized AES netlist, one byte of the AES key can be recovered. In the netlist, each S-box module contains $s = 634$ wires, indicating their belonging to the S-box with a prefix in their name. Consequently, the probability P_{2327}^* in Equation (6.1) for one S-box module is determined to be

$$P_{2327}^* = 1 - \frac{\binom{21474-634}{2327}}{\binom{21474}{2327}} = 1 - 8.2283 \times 10^{-33}$$

For a smaller percentage of inserted locking gates of only 5%, i.e. $z = 1163$, the probability P_{1163}^* would still be

$$P_{1163}^* = 1 - \frac{\binom{21474-634}{1163}}{\binom{21474}{1163}} = 1 - 2.7018 \times 10^{-16}$$

This shows that the probability of placing a locking gate at a vulnerable position inside a crypto core implementation can be significantly high. To recover all 16 bytes of the cryptographic key, at least one locking gate has to be placed within each of the 16 S-box modules. The respective probability can be calculated with the multivariate hypergeometric distribution. Further analysis is done based on one byte, thus on one faulty S-box, while it is assumed that at least one locking gate is placed at a desirable location.

6.3.2 Extracting a Secret AES Key

This section presents a template-based PFA on an AES using logic locking induced fault attacks. To demonstrate the effect of logic locking induced fault attacks on the S-box output, Figure 6.10 shows a simplified model of the final AES round. It includes the SubBytes step and the AddRoundKey step and excludes the linear ShiftRows step for simplification reasons. For each byte of an AES state, the S-box input x_j is substituted to the output y_j. Next, a roundkey byte μ_j is added to the S-box output y_j, resulting in the output c_j.

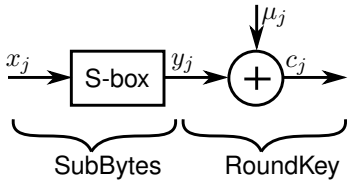


Figure 6.10 Simplified substitution layer ©2020 IEEE [Bru+20]

We cannot apply the PFA directly because the residual entropy H_{res} of the last round key is too high [Zha+18a]. The residual entropy H_{res} is the product of the number of ciphertext bytes per state L and the logarithm of the number of faulted S-box mappings λ:

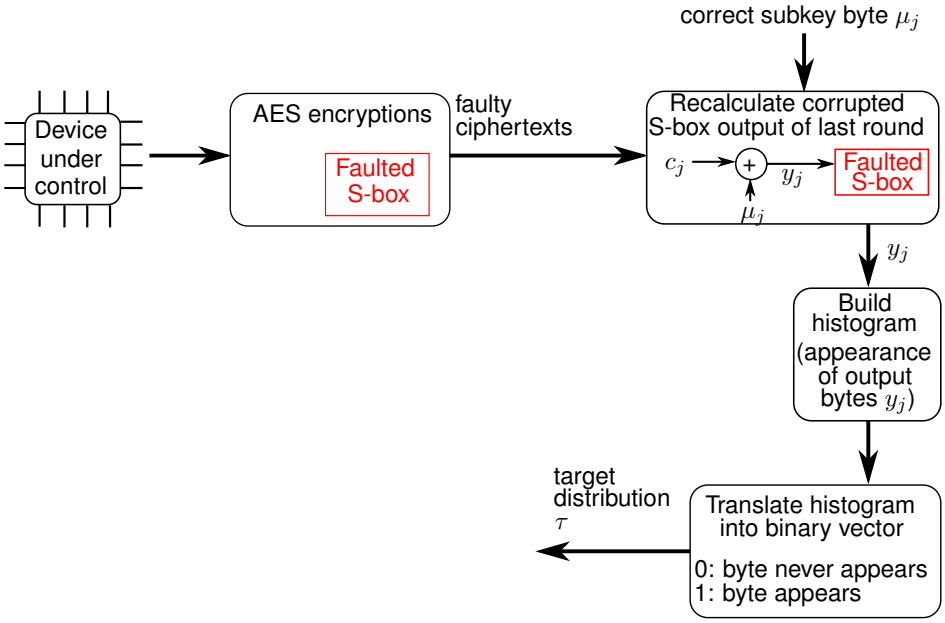
$$H_{res} = L \times \log_2(\lambda) \tag{6.2}$$

For an AES, the number of ciphertext bytes per state is 16. To determine the number of faulted S-box mappings, we analyze the S-box outputs for our faulted AES implementation based on [Uss02], see the histogram in Figure 6.12a. It shows 26 S-box outputs that no longer exist, i.e. which are for sure corrupted S-box outputs. Thus, $\lambda \geq 26$. As a result, equation (6.2) computes the minimum residual key entropy H_{res} of the last round key to be $75.2\,\mathrm{bit}$. A high minimum residual key entropy means that a brute-force complexity remains which is also high. Thus, in our use case, PFA [Zha+18a] cannot be directly applied.

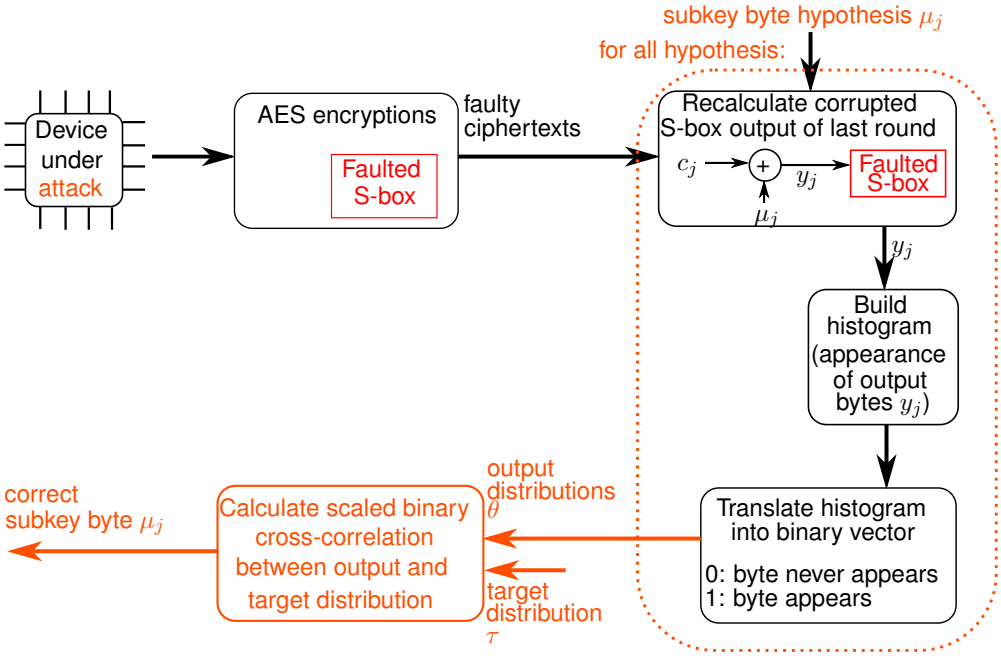
Consequently, we adapt the PFA of [Zha+18a] to make it applicable for our use case, namely logic locking induced fault injection based on the assumptions of the use case made above. This includes the same permanent fault for all devices. Due to this property, we can split the PFA into two phases, see Figure 6.11. In the first phase, the attacker generates templates based on a device under our control, i.e. the attacker can control the plaintext, ciphertext, and AES key. In the second phase, we perform the actual attack on a device under attack by modifying the locking key.

Template Phase

The complete workflow of the template phase is illustrated in Figure 6.11a. During the template phase, the attacker generates a large number of faulty ciphertexts on the device under control by corrupting the S-box substitution with logic locking induced faults. As a result, the S-box changes from a bijection to a surjection. Originally, the S-box substitution is a bijection. A bijection is distinct and can thus be reversed. When faulted, the S-box substitution changes to a surjection. A surjection is ambiguous, i.e. multiple inputs can map to the same output and can thus not be reversed. Next, the attacker can recalculate the corrupted S-box output bytes y_j of the last AES round from the observed faulty ciphertexts, as the correct AES key κ and thus the correct subkey bytes μ_j of the last AES round key are known in the template phase. The appearance of the output bytes y_j of the faulted S-box is plotted, resulting in a histogram as shown in Figure 6.12a. As mentioned before, the S-box output of the last AES round shows 26 non-existent output values. All other possible 230 output values exist. Next, the attacker has to translate the histogram into a binary vector where an existent output value is mapped to '1' and a non-existent output value is mapped to '0'. In the following, this binary vector is denoted as the target distribution τ of the S-box.

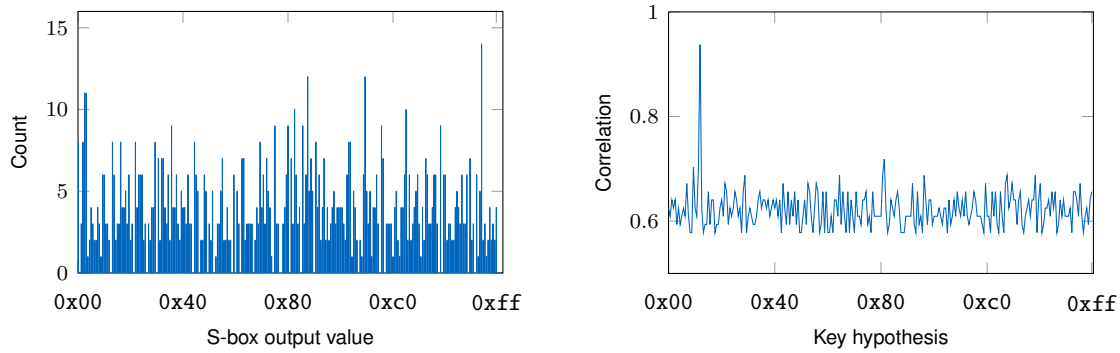


(a) Template phase



(b) Attack phase

Figure 6.11 Workflow of the two phases of the extended PFA



(a) Histogram of appearance of S-box output values

(b) Correlation of S-box output distribution θ and S-box target distribution τ

Figure 6.12 Histogram of S-box outputs and correlation diagram of AES key extraction for demonstration of logic locking induced fault attack ©2020 IEEE [Bru+20]

Attack Phase

The complete workflow of the attack phase is illustrated in Figure 6.11b. During the attack phase, we assume that the device under attack is locked with the same modified locking key as in the template phase and that the attacker can observe the faulty ciphertexts. The attacker does not have to know or control the plaintext what makes the attack a ciphertext-only attack. The first steps of the attack phase are similar to the first steps of the template phase. The attacker observes the faulty ciphertexts and uses them to calculate backward to the output of the faulted S-box. In contrast to the template phase, the attacker does not know the correct cryptographic key κ but must recalculate the S-box outputs for each possible partial subkey hypothesis μ_j, i.e. one key byte of the last round key of AES. Similar to the template phase, the resulting histograms are translated to binary vectors, now called output distributions θ. To identify the correct output distribution and thus the correct key hypothesis, the attacker compares each output distribution θ with the target distribution τ. The comparison is done by calculating a scaled binary cross-correlation using the following function:

$$\mathcal{C}(\theta, \tau) = \frac{1}{2^b} \sum_{l=0}^{2^b-1} -1^{(\theta_l + \tau_l)} \mid b = 8$$

Figure 6.12b shows the calculated cross-correlation values for each key hypothesis. It visualizes a higher cross-correlation of the correct key hypothesis and the target distribution compared to the cross-correlations of the wrong key hypothesis and the target distribution. Thus, the correct key hypothesis is successfully identified to be `0x13`. For the attack demonstration and the plots in Figure 6.12, we used 1001 faulted encryptions.

6.4 Summary

The chapter presents and analyzes a new risk of logic locking methods: logic locking induced fault attacks. We show that logic locking induced fault attacks can exploit the inserted logic locking structures to inject faults or HT without further modifying the original design. If the attacker is able to modify a locking key bit, the locked chip faces a potential new attack vector. The attack's success and applicability depend on the applied locking method, key management and storage, and the underlying fault analysis, which this work discusses. It is demonstrated that random insertion of locking gates and a permanent modification of the locking key can result

in the leakage of secret cryptographic keys. For demonstration, a modified template-based PFA is applied on an AES implementation.

In summary, as soon as a chip is used as a platform for security-relevant tasks, logic locking should not be solely seen as protection but also as a potential security risk. An insecure locking key storage and management make the produced chip and, thus, the implemented security functionality vulnerable to fault attacks, resulting in cryptographic key recovery, password bypass, or program flow changes. Locking key gates can be misused as HTs without additional effort and invisible for any HT detection method. As the HT is triggered by a modified locking key, it can be interpreted as a hardware-software solution. Overall, the integration of a vulnerable logic locking implementation and locking key management may be much worse than producing designs without logic locking at all.

7 Conclusion and Summary

This thesis contributes to a new perspective in sequential RE and obfuscation. Summarizing from Chapter 1, the contributions of the thesis are as follows: It derives a novel categorization for sFF sets and an sFF set definition for human-readable state machines by analyzing features of sFFs and sFF sets, see Chapter 3. It develops post-processing approaches to improve existing sFF identification methods, see Chapter 4, and it develops two novel FSM obfuscation techniques to prevent the successful RE of FSMs, see Chapter 5. On top, the thesis introduces logic locking induced fault attacks by injecting faults with flipping single bits of the logic locking key, see Chapter 6.

Two topics span the contributions of this thesis:

1. Exploring features of hardware gate-level netlists.
2. Combining different techniques.

Figure 7.1 shows how the contributions of this thesis relate to these two statements. The following sections elaborate on both in more detail.

7.1 Exploring Features of Hardware Gate-Level Netlists

A hardware gate-level netlist represents the design's functionality. It is the output when synthesizing the design's RTL description [Utm21], see Section 2.1. Depending on the design, the gate-level netlist holds different features, including structural or functional features. By exploring these features, one can achieve a better understanding of how specific design functionalities are described by a gate-level netlist. This can have the following effects:

- One can use this knowledge to improve RE methods. Functional RE analyses hardware gate-level netlists to extract high-level information and gain a better understanding of a design, see Section 2.3. The gate-level netlist analysis uses functional and structural features to predict the coherence or the functionality of single components in the gate-level netlist, e.g. gates or wires, or of component groups, e.g. netlist subgraphs [Azr+21]. The better one understands the relation between gate-level netlist features and the design's functionality, the more suitable features can be chosen for functional RE algorithms what improves the overall RE results.
- One can use the knowledge to improve obfuscation techniques. If a design does not show functional or structural features typical of it, RE algorithms are weakened or fail. Thus, describing designs without such typical functional or structural features obfuscates the designs against RE methods. Furthermore, other obfuscation techniques are only feasible if a design fulfills specific structural features. Thus, describing designs with specific structural features allows the applicability of certain obfuscation techniques.
- One can use the knowledge to improve the understanding of attack capabilities beside RE. If one can map the attack capabilities on certain types of netlist structures, one can predict the attack capabilities without performing the attack itself but based on a theoretical analysis.

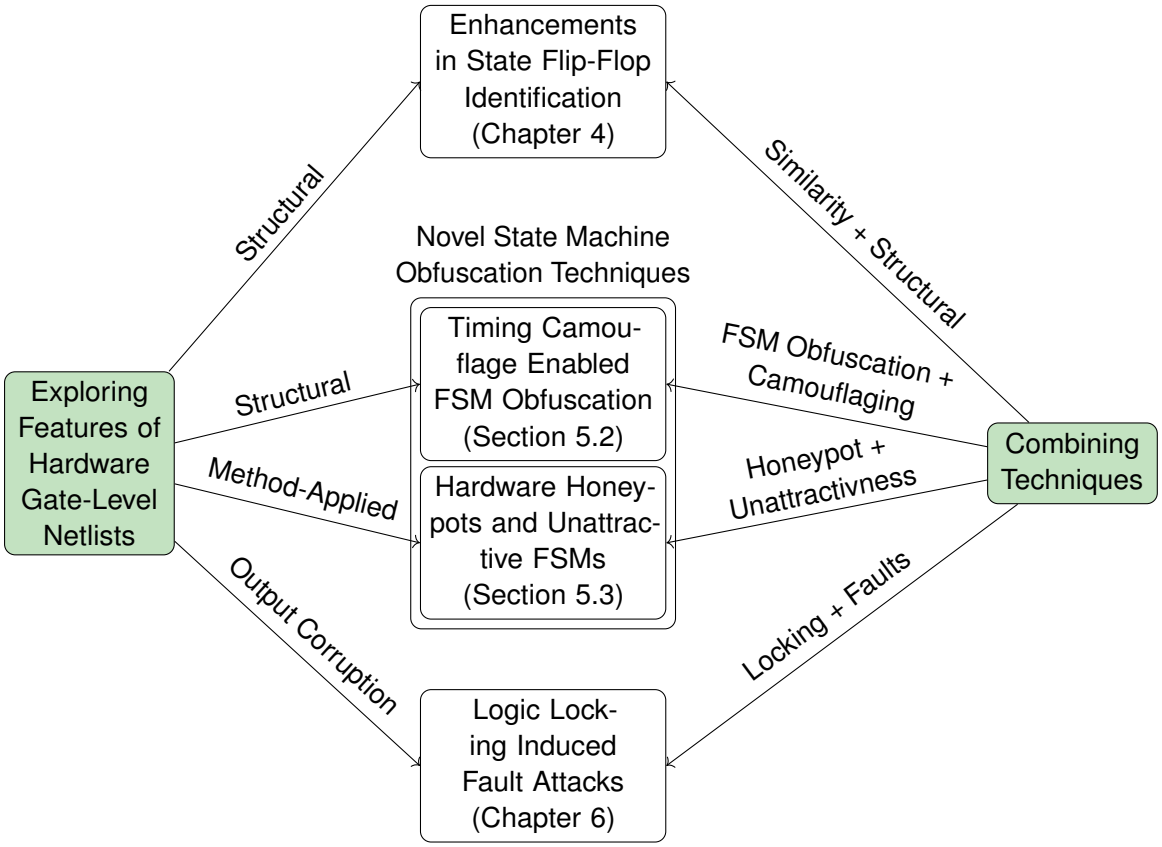


Figure 7.1 Structure of the thesis and its relation to the topics "Exploring Features of Hardware Gate-Level Netlists" and "Combining Techniques"

In particular, for FSMs, the feature exploration of gate-level netlists is of special interest because FSMs can show significant differences in the netlist structure compared to data-driven circuit components [Ped13; Azr+21], see Chapter 1. This thesis contributes to the feature analysis of FSMs and locking techniques and analyzes the applied features of existing RE methods, see Chapters 3, 4, and 6. Chapters 4, 5, and 6 make use of the explored features in a gate-level netlist, see Figure 7.1:

- The enhancements in the sFF identification (Chapter 4) consider the following three observed *structural* features of sFFs in a gate-level netlist: connectivity, path class, and condition of membership. Based on these features, we implement a post-processing method that adds and/or removes FFs from the potential sFF set. The post-processing can consider various values of these features. All feature values are assumed to represent sFFs; however, some include more, and some include fewer types of various FSM implementations.

- The concepts of both novel FSM obfuscation techniques are based on the explored gate-level netlist features. FSM-TC (Section 5.2) makes use of the following *structural* feature: combinational FP. Timing Camouflage can be applied on sFFs because FSMs are redesigned such that they no longer have a combinational FP. Instead, the obfuscation technique of hardware honeypots and unattractive FSMs (Section 5.3) makes use of the features which are *applied by RE methods*. The obfuscation technique designs honeypot FSMs, which fulfill many of these features, and redesigns the original FSMs into unattractive FSMs by removing features.

- The applicability of logic locking induced fault attacks (Chapter 6) is derived by analyzing the *output corruption behavior* of logic locking techniques. Logic locking induced fault attacks achieve the best applicability for a *flip* output corruption, i.e. flipping a locking key bit results in flipping the original circuit signal. The analysis enables a prediction about the attack's capability without the need to perform the attack for all possible locking methods.

7.2 Combining Techniques

The idea of combining different techniques to gain an advantage or to improve or strengthen processes or methods is not new. For example, early locking methods suggest combining their SAT-resistant locking technique with the existing XOR/XNOR locking method [Yas+16a; Yas+16b]. Also, Rahman et al. [Rah+20b] present a holistic approach that suggests the combination of various obfuscation techniques to prevent multiple attack scenarios. Each obfuscation technique targets one or more specific attack scenarios, resulting in an overall protection of the design. Besides this well-known fact that the combination of techniques can improve protection approaches, the combination of techniques can also improve attack approaches. In particular, in the context of RE, many approaches exist which combine various techniques to reverse engineer a netlist [e.g. Fyr+18; Sub+14; Azr+21].

This thesis presents benefits but also risks of combining techniques in the context of sequential RE, locking, and FSM obfuscation, see Figure 7.1:

- To enhance the sFF identification (Chapter 4), we developed post-processing algorithms. The post-processing algorithms add and/or remove FFs from a potential sFF set, using structural features. The potential sFF set is the output of an existing sFF identification technique, preferably one that uses similarity-based features. Thus, when applying structure-driven post-processing on the output of a similarity-driven sFF identification method, we combine *similarity features with structural features*. The results largely show an improved success of identifying sFFs.

- The two novel obfuscation methods both use a combination of different techniques. FSM-TC (Section 5.2) combines the *camouflaging technique, Timing Camouflage, with the obfuscation of FSMs*. It applies Timing Camouflage on sFFs, which removes them in a reverse-engineered gate-level netlist and thus complicates the FSM extraction significantly. Instead, the obfuscation technique of hardware honeypots and unattractive FSMs (Section 5.3) combines, as the name suggests, *attractive honeypot FSMs and unattractive FSMs*. While the honeypots are added as extra FSMs to the designs and are highly attractive to RE tools, original FSMs are obfuscated, resulting in unattractive FSMs.

- To insert logic locking induced faults (Chapter 6), the attacker uses the inserted gates and the locking key input of logic locking methods. By flipping one bit of the locking key, the functionality of the circuit is corrupted, i.e. a signal value is flipped, for at least one input pattern. Thus, the attack combines the topics *fault injection and analysis and logic locking techniques*. In addition, the analysis of logic locking induced faults uncovers another risk of combining methods: If logic locking methods are combined independently, i.e. each method uses an independent, separate locking key, the attacker can choose the more suitable locking method for applying logic locking induced fault attacks. Thus, a poor combination of techniques might not only have no effect but even a negative effect on the security.

7.3 Impact and Future Work

The impact of hardware RE in general and of sequential RE in particular is discussed in Chapter 1 in detail. This includes a high interest for attackers to steal the design IP or to gain knowledge about possible attack targets. This also includes a high interest for chip owners to verify their manufactured product against malicious modifications.

This thesis contributes to the improvement of sequential RE and FSM obfuscation.

It shows how the analysis of features of gate-level netlists and sFF identification methods can advance strategies of identifying sFFs or can lead to novel, innovative obfuscation strategies, see Section 7.1. Understanding the theoretical nature of the attack or obfuscation target can have a crucial impact on the success of the attack or obfuscation. Similarly, for machine learning applications, understanding the features of the target enables a justified parameter selection and thus an improvement of the machine learning results. Machine learning is a rising area and has also already been applied for sequential RE [CYN21; Alr+22], what as well underlines the significance of understanding the target's features. Also, with continuous improvement of sequential RE techniques and attacks, existing obfuscation techniques must be regularly checked for validity and, if necessary, revised and improved. In particular, the second novel FSM obfuscation method, hardware honeypots and unattractive FSM in Section 5.3, has the potential to be further developed so that it is still secure against future sequential RE techniques.

Additionally, the thesis once again supports the hypothesis that combining different protection techniques can improve or, in this case, also enable new protection mechanisms, see Section 7.2. The thesis also supports the hypothesis that combining different attack techniques can also strengthen attack possibilities. In addition, the thesis provides a new perspective: Combining protection techniques can also ease the applicability of attacks, as is the case for logic locking induced fault attacks. The benefit of combining techniques has long been used in various other research fields, like in hardware/software co-design [Ha+17] or in big.LITTLE technology of ARM [ARM13] to improve performance. There is more potential to further improve or develop new protection mechanisms and RE strategies by combining various techniques. Each of the combined techniques can act as a single component or reinforce each other. Besides this positive effect, the thesis, however, also demonstrated the potential risk of combinations. Thus, future research that uses combinations of techniques should also always check its method's potential to enable or ease attacks due to an improperly implemented combination technique.

A Appendix

A.1 Post-Processing Results

FN and FP values for the remaining three post-processing methods which are not considered in Section 4.2.4, Table 4.7.

Table A.1 FN and FP based on $\mathbb{F}_{\mathsf{H}}^{s}$ for a similarity-based identification method, fastRELIC, with the post-processing methods, PP - weak,generic, PP - weak,FSM, and PP - strong,generic, using benchmarks of Table 4.1.

Design	PP - weak,generic		PP - weak,FSM		PP - strong,generic	
	FN	FP	FN	FP	FN	FP
α)	0	0	0	0	0	0
β)	0	0	0	0	0	0
γ)	0	0	0	0	0	0
δ)	0	2	0	2	1	1
ϵ)	0	8	0	8	0	7
ζ)	0	4	0	4	0	3
η)	0	5	0	4	0	3
θ)	0	16	0	16	0	15
ι)	0	16	0	16	3	12
κ)	0	2	0	2	0	0
λ)	0	1	0	1	0	0
μ)	0	0	0	0	0	0
ν)	0	31	0	27	0	23
o)	0	5	0	0	0	0
π)	3	10	3	8	3	4
ρ)	0	42	0	32	3	36
σ)	9	17	7	16	9	4
τ)	51	97	50	99	51	87
ϕ)	1	13	0	13	6	12
χ)	0	11	0	9	0	8
ψ)	0	24	0	24	4	21

A.2 Pseudocode for Use Case Example Design

The pseudocode in Algorithm 4 is used as an example for a large size discrepancy of $\mathbb{F}_{\mathsf{H}}^{s}$ and its corresponding FSM SCC. The $\mathbb{F}_{\mathsf{H}}^{s}$ consists of two FFs which represent the register *stato* after synthesis. The corresponding FSM SCC consists of 19 FFs, which represent the registers *stato*, *d_in*, and *old* after synthesis.

Algorithm 4 Simplified pseudocode excerpt of an example design of the ITC99 benchmarks with an added reset, *b09_reset* [CRS00]. The register *stato* is colored in green and its corresponding FFs form $\mathbb{F}_H^s$, while the two registers, *d_in* and *old* are colored in blue and their corresponding FFs additionally belong to the corresponding FSM SCC.

```
Data: reset, clock, x
Result: y - not considered here
...
always @(posedge clock)
    if !reset then
        stato = Load_old
        d_in = '000000000'
        old = '00000000'
    else
        switch stato do
            case Init
                stato = Receive
                d_in = '000000000'
                old = '00000000'
            case Receive
                if d_in[0] = '1' then
                    old = d_in[8:1]
                    d_in = '100000000'
                    stato = Execute
                else
                    d_in = x & d_in[8:1]
                    stato = Receive
            case Execute
                if d_in[0] = '1' then
                    stato = Load_old
                else
                    stato = Execute
                d_in = x & d_in[8:1]
            case Load_old
                if d_in[0] = '1' then
                    if d_in[8:1] = old then
                        d_in = '000000000'
                        stato = Load_old
                    else
                        d_in = '100000000'
                        stato = Execute
                    old = d_in[8:1]
                else
                    d_in = x & d_in[8:1]
                    stato = Load_old
```

List of Figures

List of Tables

List of Algorithms

Acronyms

AES Advanced Encryption Standard

ASIC Application Specific Integrated Circuit

ASSP Application Specific Standard Product

CFA Collision Fault Analysis

CLB Complex Logic Block

EDA Electronic Design Automation

FP False Positive

FN False Negative

FF Flip-Flop

FF set Flip-Flop Set

FP Feedback Path

FPGA Field-Programmable-Gate-Array

FSM Finite State Machine

FSM-HP Finite State Machine Honeypot

FSM-TC Timing Camouflage Enabled FSM Obfuscation

GNN Graph Neural Network

HDL Hardware Description Language

HT Hardware Trojan

IC Integrated Circuit

IP Intellectual Property

LFSR Linear-Feedback Shift Register

LUT Lookup-Table

nFF Non State Flip-Flop

nFF set Non State Flip-Flop Set

NLFSR Nonlinear-Feedback Shift Register

NN Neural Network

OSU Oklahoma State University

OWF One-Way Function

PCA Principal Component Analysis

PCB Printed Circuit Board

PFA Persistent Fault Analysis

PUF Physical Unclonable Function

RE Reverse Engineering

RTL Register Transfer Level

SAT Satisfiability

SCC Strongly Connected Component

SEM Scanning Electron Microscope

sFF State Flip-Flop

sFF set State Flip-Flop Set

SMT Satisfiability Modulo Theories

SoC System on Chip

STG State Transition Graph

STT State Transition Table

TC Timing Camouflage

Nomenclature

Finite State Machine

$\mathcal{M}$	Mathematical description of a Finite State Machine
δ	Transition function
λ	Output function
$\mathbb{S}$	Finite set of states
S^*	Initial state in $\mathbb{S}$
S_i	State in $\mathbb{S}$
m	Number of states in $\mathbb{S}$
$\mathbb{I}$	Finite set of inputs
I_l	Input in $\mathbb{I}$
i_l^t	Value of I_l at time t
k	Number of inputs in $\mathbb{I}$
$\mathbb{O}$	Finite set of outputs
$\mathbb{E}(S_i, S_j)$	Set of transition conditions from S_i to S_j
$e_l(S_i, S_j)$	Transition condition in $\mathbb{E}(S_i, S_j)$
q	Number of transition conditions in $\mathbb{E}(S_i, S_j)$

Gate-Level Netlist

$\mathbb{F}$	Finite Flip-Flop Set
F_j	Flip-Flop in $\mathbb{F}$
$\mathbb{P}$	Finite set of subsets of a Flip-Flop Set
$\mathbb{F}^s$	Finite State Flip-Flop Set
F_j^s	State Flip-Flop in $\mathbb{F}^s$
$f_j^{s,t}$	Value of F_j^s at time t
n	Number of State Flip-Flops for a Finite State Machine
$\mathbb{F}_{\mathsf{H}}^s$	State Flip-Flop Set for a human-readable Finite State Machine definition
$\mathbb{F}^n$	Finite Non State Flip-Flop Set

F_j^n Non State Flip-Flop in $\mathbb{F}^n$

u Number of Non State Flip-Flops for a Finite State Machine

$\mathbb{N}$ Finite set of combinational gates

g_l Gate in $\mathbb{N}$ or $\mathbb{F}$

w Number of wires

s Number of wires which are a suitable place for a successful fault injection attack

$p(g_1, g_l)$ Connected sequence of gates $(g_1, \ldots, g_l)$

$P(g_1, g_l)$ Function whose output is true if a path $p(g_1, g_l)$ exists

Locking and Encryption

K Logic locking key

z Number of inserted locking key gates

κ Cryptographic key

κ_j Byte j of cryptographic key κ

μ Extended AES master key

μ_j Byte j of extended AES master key μ

c_j Byte j of AES ciphertext

x_j Byte j of AES S-box input

y_j Byte j of AES S-box output

L Number of ciphertext bytes per state

λ Number of faulted S-box mappings

θ Output distribution of faulty S-box output value appearances for key hypothesis

τ Target distribution of faulty S-box output value appearances for correct key

Other symbols

o Obfuscation signal

L_f List of all not paired transitions after transition pair check for Redesign_Func

$P(\cdot)$ Hypergeometric probability

P_z^* Probability that at least one suitable wire is locked when inserting z locking gates

H_{res} Residual entropy

$\mathcal{C}(\cdot, \cdot)$ Scaled binary cross-correlation

Bibliography

[ADN91] P. Ashar, S. Devadas, and A.R. Newton. “Optimum and heuristic algorithms for an approach to finite state machine decomposition”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.3 (1991), pp. 296–310. DOI: 10.1109/43.67784.

[AGH19] Mahabubul Alam, Swaroop Ghosh, and Sujay S. Hosur. “TOIC: Timing Obfuscated Integrated Circuits”. In: *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. GLSVLSI ’19. Tysons Corner, VA, USA: Association for Computing Machinery, 2019, pp. 105–110. ISBN: 9781450362528. DOI: 10.1145/3299874.3318001. https://doi.org/10.1145/3299874.3318001.

[AGM19] Leonid Azriel, Ran Ginosar, and Avi Mendelson. “SoK: An Overview of Algorithmic Methods in IC Reverse Engineering”. In: *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*. ASHES’19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 65–74. ISBN: 9781450368391. DOI: 10.1145/3338508.3359575. https://doi.org/10.1145/3338508.3359575.

[AK07] Yousra M. Alkabani and Farinaz Koushanfar. “Active Hardware Metering for Intellectual Property Protection and Security”. In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. SS’07. Boston, MA: USENIX Association, 2007. ISBN: 1113335555779.

[AKP07] Yousra Alkabani, Farinaz Koushanfar, and Miodrag Potkonjak. “Remote activation of ICs for piracy prevention and digital right management”. In: *2007 IEEE/ACM International Conference on Computer-Aided Design*. 2007, pp. 674–677. DOI: 10.1109/ICCAD.2007.4397343.

[Alb+20] Nils Albartus, Max Hoffmann, Sebastian Temme, Leonid Azriel, and Christof Paar. “DANA Universal Dataflow Analysis for Gate-Level Netlist Reverse Engineering”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.4 (Aug. 1, 2020), pp. 309–336. DOI: 10.13154/tches.v2020.i4.309-336. https://tches.iacr.org/index.php/TCHES/article/view/8685. published.

[Alr+22] Lilas Alrahis et al. “GNN-RE: Graph Neural Networks for Reverse Engineering of Gate-Level Netlists”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41.8 (2022), pp. 2435–2448. DOI: 10.1109/TCAD.2021.3110807.

[ARM13] ARM. *big.LITTLE Technology: The Future of Mobile*. Accessed on November 2023. 2013. https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/big-little-technology-the-future-of-mobile.pdf.

[Aza+18] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. “SMT Attack: Next Generation Attack on Obfuscated Circuits with Capabilities and Performance Beyond the SAT Attacks”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.1 (Nov. 2018), pp. 97–122. DOI: 10.13154/tches.v2019.i1.97-122. https://tches.iacr.org/index.php/TCHES/article/view/7335.

[Aza+19] Kimia Zamiri Azar et al. “COMA: Communication and Obfuscation Management Architecture”. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 181–195. ISBN: 978-1-939133-07-6. https://www.usenix.org/conference/raid2019/presentation/azar.

[Azr+16] Leonid Azriel, Ran Ginosar, Shay Gueron, and Avi Mendelson. “Using Scan Side Channel for Detecting IP Theft”. In: HASP 2016 (2016). DOI: 10.1145/2948618.2948619. https://doi.org/10.1145/2948618.2948619.

[Azr+21] Leonid Azriel, Julian Speith, Nils Albartus, Ran Ginosar, Avi Mendelson, and Christof Paar. “A survey of algorithmic methods in IC reverse engineering”. In: *Journal of Cryptographic Engineering* 11.3 (2021), pp. 299–315. DOI: 10.1007/s13389-021-00268-5. https://doi.org/10.1007/s13389-021-00268-5.

[Bae+19] Johanna Baehr, Alessandro Bernardini, Georg Sigl, and Ulf Schlichtmann. “Machine Learning and Structural Characteristics for Reverse Engineering”. In: ASPDAC ’19 (2019), pp. 96–103. DOI: 10.1145/3287624.3288740. https://doi.org/10.1145/3287624.3288740.

[Bae+22] Johanna Baehr, Alexander Hepp, Michaela Brunner, Maja Malenko, and Georg Sigl. “Open Source Hardware Design and Hardware Reverse Engineering: A Security Analysis”. In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. 2022, pp. 504–512. DOI: 10.1109/DSD57027.2022.00073.

[Bar+12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. “Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures”. In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769.

[BBS19] Michaela Brunner, Johanna Baehr, and Georg Sigl. “Improving on State Register Identification in Sequential Hardware Reverse Engineering”. In: *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2019, pp. 151–160. DOI: 10.1109/HST.2019.8740844.

[BGV15] Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. “Electromagnetic circuit fingerprints for Hardware Trojan detection”. In: *2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*. 2015, pp. 246–251. DOI: 10.1109/ISEMC.2015.7256167.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. “Gephi: An Open Source Software for Exploring and Manipulating Networks”. In: (2009). http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

[BK06] Johannes Blömer and Volker Krummel. “Fault Based Collision Attacks on AES”. In: *Fault Diagnosis and Tolerance in Cryptography*. Ed. by Luca Breveglieri, Israel Koren, David Naccache, and Jean-Pierre Seifert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 106–120. ISBN: 978-3-540-46251-4.

[Bru+20] Michaela Brunner, Michael Gruber, Michael Tempelmeier, and Georg Sigl. “Logic Locking Induced Fault Attacks”. In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 114–119. DOI: 10.1109/ISVLSI49217.2020.00030.

[Bru+22a] Michaela Brunner, Alexander Hepp, Johanna Baehr, and Georg Sigl. “Toward a Human-Readable State Machine Extraction”. In: *ACM Trans. Des. Autom. Electron. Syst.* 27.6 (June 2022). ISSN: 1084-4309. DOI: 10.1145/3513086. https://doi.org/10.1145/3513086.

[Bru+22b] Michaela Brunner, Tarik Ibrahimpasic, Bing Li, Grace Li Zhang, Ulf Schlichtmann, and Georg Sigl. “Timing Camouflage Enabled State Machine Obfuscation”. In: *2022 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 2022, pp. 1–7. DOI: 10.1109/PAINE56030.2022.10014810.

[Bru+23] Michaela Brunner, Hye Hyun Lee, Alexander Hepp, Johanna Baehr, and Georg Sigl. “Hardware Honeypot: Setting Sequential Reverse Engineering on a Wrong Track”. In: *arXiv preprint arXiv:2305.03707* (2023). Peer-reviewed version: *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*. 2024, pp. 47–52. DOI:10.1109/DDECS60919.2024.10508924.

[Bru17] Michaela Brunner. *Sequentiel Netlist Reverse Engineering*. Research Internship, Evaluated on 16th November 2017. 2017.

[Bru18] Michaela Brunner. *Improving Sequential Gate-level Netlist Reverse Engineering and Obfuscation*. Master Thesis, Evaluated on 5th June 2018. 2018.

[BTZ10] Alex Baumgarten, Akhilesh Tyagi, and Joseph Zambreno. “Preventing IC Piracy Using Reconfigurable Logic Barriers”. In: *IEEE Design & Test of Computers* 27.1 (2010), pp. 66–75. DOI: 10.1109/MDT.2010.24.

[Bur+98] W.P. Burleson, M. Ciesielski, F. Klass, and W. Liu. “Wave-pipelining: a tutorial and research survey”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6.3 (1998), pp. 464–474. DOI: 10.1109/92.711317.

[CB09] Rajat Subhra Chakraborty and Swarup Bhunia. “HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.10 (2009), pp. 1493–1502. DOI: 10.1109/TCAD.2009.2028166.

[CBC07] Lap-Wai Chow, James P. Baukus, and William M. Clark. *Integrated circuits protected against reverse engineering and method for fabricating the same using an apparent metal contact line terminating on field oxide*. US Patent No. US 7,294,935 B2, Filed Jan. 24th., 2001, Issued Nov. 13th., 2007. Nov. 2007.

[CDC11] Gerardo Canfora, Massimiliano Di Penta, and Luigi Cerulo. “Achievements and Challenges in Software Reverse Engineering”. In: *Commun. ACM* 54.4 (Apr. 2011), pp. 142–151. ISSN: 0001-0782. DOI: 10.1145/1924421.1924451. https://doi.org/10.1145/1924421.1924451.

[CFT21] Muhtadi Choudhury, Domenic Forte, and Shahin Tajik. “PATRON: A Pragmatic Approach for Encoding Laser Fault Injection Resistant FSMs”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 569–574. DOI: 10.23919/DATE51398.2021.9474222.

[Che+22] Chi-Wei Chen, Pei-Yu Lo, Wei-Ting Hsu, Chih-Wei Chen, Chin-Wei Tien, and Sy-Yen Kuo. “A Hardware Trojan Insertion Framework against Gate-Level Netlist Structural Feature-based and SCOAP-based Detection”. In: *2022 IEEE 65th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2022, pp. 1–5. DOI: 10.1109/MWSCAS54063.2022.9859423.

[CNB13] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. *Protection of intellectual property cores through a design flow*. US Patent No. US 8402,401 B2, Filed Nov. 9th., 2010, Issued Mar. 19th., 2013. Mar. 2013.

[Col+22] Luca Collini, Benjamin Tan, Christian Pilato, and Ramesh Karri. “Reconfigurable Logic for Hardware IP Protection: Opportunities and Challenges (Invited Paper)”. In: *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2022, pp. 1–7.

[Cot69] L. W. Cotten. “Maximum-Rate Pipeline Systems”. In: *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference*. AFIPS ’69 (Spring). Boston, Massachusetts: Association for Computing Machinery, 1969, pp. 581–586. ISBN: 9781450379021. DOI: 10.1145/1476793.1476883. https://doi.org/10.1145/1476793.1476883.

[Cou+16] Jacob Couch, Elizabeth Reilly, Morgan Schuyler, and Bradley Barrett. “Functional block identification in circuit design recovery”. In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2016, pp. 75–78. DOI: 10.1109/HST.2016.7495560.

[CRS00] F. Corno, M.S. Reorda, and G. Squillero. “RT-level ITC’99 benchmarks and first ATPG results”. In: *IEEE Design & Test of Computers* 17.3 (2000). Git: https://github.com/squillero/itc99-poli, pp. 44–53. DOI: 10.1109/54.867894.

[CS22] Animesh Chhotaray and Thomas Shrimpton. “Hardening Circuit-Design IP Against Reverse-Engineering Attacks”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, pp. 1672–1689. DOI: 10.1109/SP46214.2022.9833634.

[CT05] Hamid Choukri and Michael Tunstall. “Round reduction using faults”. In: *FDTC* 5 (2005), pp. 13–24.

[CYN21] Subhajit Dutta Chowdhury, Kaixin Yang, and Pierluigi Nuzzo. “ReIGNN: State Register Identification Using Graph Neural Networks for Circuit Reverse Engineering”. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643498.

[CYN23] Subhajit Dutta Chowdhury, Kaixin Yang, and Pierluigi Nuzzo. “SimLL: Similarity-Based Logic Locking Against Machine Learning Attacks”. In: *2023 60th ACM/IEEE Design Automation Conference (DAC)*. 2023, pp. 1–6. DOI: 10.1109/DAC56929.2023.10247822.

[Des+13] Avinash R. Desai, Michael S. Hsiao, Chao Wang, Leyla Nazhandali, and Simin Hall. “Interlocking Obfuscation for Anti-Tamper Hardware”. In: *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*. CSIIRW ’13. Oak Ridge, Tennessee, USA: Association for Computing Machinery, 2013. ISBN: 9781450316873. DOI: 10.1145/2459976.2459985. https://doi.org/10.1145/2459976.2459985.

[DN88] S. Devadas and A.R. Newton. “Decomposition and factorization of sequential finite state machines”. In: *[1988] IEEE International Conference on Computer-Aided Design (ICCAD-89) Digest of Technical Papers*. 1988, pp. 148–151. DOI: 10.1109/ICCAD.1988.122482.

[DQ14] Carson Dunbar and Gang Qu. “Designing Trusted Embedded Systems from Finite State Machines”. In: *ACM Trans. Embed. Comput. Syst.* 13.5s (Oct. 2014). ISSN: 1539-9087. DOI: 10.1145/2638555. https://doi.org/10.1145/2638555.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. en. Information Security and Cryptography. Berlin, Heidelberg: Springer-Verlag, 2002. DOI: 10.1007/978-3-662-04722-4. http://link.springer.com/10.1007/978-3-662-04722-4.

[Dup+14] Sophie Dupuis, Papa-Sidi Ba, Giorgio Di Natale, Marie-Lise Flottes, and Bruno Rouzeyre. “A novel hardware logic encryption technique for thwarting illegal overproduction and Hardware Trojans”. In: *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*. 2014, pp. 49–54. DOI: 10.1109/IOLTS.2014.6873671.

[DY18] Jaya Dofe and Qiaoyan Yu. “Novel Dynamic State-Deflection Method for Gate-Level Design Obfuscation”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.2 (2018), pp. 273–285. DOI: 10.1109/TCAD.2017.2697960.

[EHP19] Susanne Engels, Max Hoffmann, and Christof Paar. *The End of Logic Locking? A Critical View on the Security of Logic Locking*. Cryptology ePrint Archive, Paper 2019/796. 2019. https://eprint.iacr.org/2019/796.

[Emb19] Embedded Security Group. *HAL - The Hardware Analyzer*. https://github.com/emsec/hal. 2019.

[Fyr+17] Marc Fyrbiak et al. “Hardware reverse engineering: Overview and open challenges”. In: *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*. 2017, pp. 88–94. DOI: 10.1109/IVSW.2017.8031550.

[Fyr+18] Marc Fyrbiak et al. “On the Difficulty of FSM-based Hardware Obfuscation”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (Aug. 2018), pp. 293–330. DOI: 10.13154/tches.v2018.i3.293-330. https://tches.iacr.org/index.php/TCHES/article/view/7277.

[Gei+20] James Geist, Travis Meade, Shaojie Zhang, and Yier Jin. “RELIC-FUN: Logic Identification through Functional Signal Comparisons”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218616.

[Gei+23] Jim Geist, Travis Meade, Shaojie Zhang, and Yier Jin. “Improving FSM State Enumeration Performance for Hardware Security with RECUT and REFSM-SAT”. In: *arXiv preprint arXiv:2311.10273* (2023).

[GRR12] W. Paul Griffin, Anand Raghunathan, and Kaushik Roy. “CLIP: Circuit Level IC Protection Through Direct Injection of Process Variations”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 20.5 (2012), pp. 791–803. DOI: 10.1109/TVLSI.2011.2135868.

[Gui+16] Ujjwal Guin, Qihang Shi, Domenic Forte, and Mark M. Tehranipoor. “FORTIS: A Comprehensive Solution for Establishing Forward Trust for Protecting IPs and ICs”. In: *ACM Trans. Des. Autom. Electron. Syst.* 21.4 (May 2016). ISSN: 1084-4309. DOI: 10.1145/2893183. https://doi.org/10.1145/2893183.

[GZS18] Ujjwal Guin, Ziqi Zhou, and Adit Singh. “Robust Design-for-Security Architecture for Enabling Trust in IC Manufacturing and Test”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.5 (2018), pp. 818–830. DOI: 10.1109/TVLSI.2018.2797019.

[Ha+17] Soonhoi Ha et al. “Introduction to Hardware/Software Codesign”. In: *Handbook of Hardware/Software Codesign*. Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 1–24. ISBN: 978-94-017-7358-4. DOI: 10.1007/978-94-017-7358-4_41-1. https://doi.org/10.1007/978-94-017-7358-4_41-1.

[Har60] J. Hartmanis. “Symbolic analysis of a decomposition of information processing machines”. In: *Information and Control* 3.2 (1960), pp. 154–178. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(60)90744-0. https://www.sciencedirect.com/science/article/pii/S0019995860907440.

[HBS22] Alexander Hepp, Johanna Baehr, and Georg Sigl. “Golden Model-Free Hardware Trojan Detection by Classification of Netlist Module Graphs”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 1317–1322. DOI: 10.23919/DATE54114.2022.9774760.

[Hon+22] Xuenong Hong, Tong Lin, Yiqiong Shi, and Bah Hwee Gwee. “GraphClusNet: A Hierarchical Graph Neural Network for Recovered Circuit Netlist Partitioning”. In: *IEEE Transactions on Artificial Intelligence* (2022), pp. 1–15. DOI: 10.1109/TAI.2022.3198930.

[HP20] Max Hoffmann and Christof Paar. “Doppelganger Obfuscation — Exploring theDefensive and Offensive Aspects of Hardware Camouflaging”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (Oct. 2020), pp. 82–108. DOI: 10.46586/tches.v2021.i1.82-108. https://tches.iacr.org/index.php/TCHES/article/view/8728.

[Hu+19] Wei Hu, Yixin Ma, Xinmu Wang, and Xingxin Wang. “Leveraging Unspecified Functionality in Obfuscated Hardware for Trojan and Fault Attacks”. In: *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. 2019, pp. 1–6. DOI: 10.1109/AsianHOST47458.2019.9006725.

[Hu+20] Yinghua Hu, Kaixin Yang, Shahin Nazarian, and Pierluigi Nuzzo. “SANSCrypt: A Sporadic-Authentication-Based Sequential Logic Encryption Scheme”. In: *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. 2020, pp. 129–134. DOI: 10.1109/VLSI-SOC46417.2020.9344079.

[Hu+21] Yinghua Hu, Yuke Zhang, Kaixin Yang, Dake Chen, Peter A. Beerel, and Pierluigi Nuzzo. “Fun-SAT: Functional Corruptibility-Guided SAT-Based Attack on Sequential Logic Encryption”. In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2021, pp. 281–291. DOI: 10.1109/HOST49136.2021.9702267.

[HYH99] M.C. Hansen, H. Yalcin, and J.P. Hayes. “Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering”. In: *IEEE Design & Test of Computers* 16.3 (1999), pp. 72–80. DOI: 10.1109/54.785838.

[JS18] Kyle Juretus and Ioannis Savidis. “Time Domain Sequential Locking for Increased Security”. In: *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351412.

[JS21a] Kyle Juretus and Ioannis Savidis. “Synthesis of Hidden State Transitions for Sequential Logic Locking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.1 (2021), pp. 11–23. DOI: 10.1109/TCAD.2020.2994259.

[JS21b] Kyle Joseph Juretus and Ioannis Savidis. *Enhanced circuity security through hidden state transitions*. US Patent App. 16/980,471. Jan. 2021.

[Kam+19] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. “Full-Lock: Hard Distributions of SAT instances for Obfuscating Circuits using Fully Configurable Logic and Routing Blocks”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[Kam+20] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. “InterLock: An Intercorrelated Logic and Routing Locking”. In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2020, pp. 1–9.

[Kam+22] Hadi Mardani Kamali, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. *Advances in Logic Locking: Past, Present, and Prospects*. Cryptology ePrint Archive, Paper 2022/260. 2022. https://eprint.iacr.org/2022/260.

[KCK20] Rajit Karmakar, Santanu Chattopadhyay, and Rohit Kapur. “A Scan Obfuscation Guided Design-for-Security Approach for Sequential Circuits”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 67.3 (2020), pp. 546–550. DOI: 10.1109/TCSII.2019.2915606.

[Kib+22] Rasheed Kibria, Nusrat Farzana, Farimah Farahmandi, and Mark Tehranipoor. “FSMx: Finite State Machine Extraction from Flattened Netlist With Application to Security”. In: *2022 IEEE 40th VLSI Test Symposium (VTS)*. 2022, pp. 1–7. DOI: 10.1109/VTS52500.2021.9794151.

[KJC19] Rajit Karmakar, Suman Sekhar Jana, and Santanu Chattopadhyay. “A Cellular Automata Guided Obfuscation Strategy For Finite-State-Machine Synthesis”. In: *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 2019, pp. 1–6.

[KMV20] Yasaswy Kasarabada, Vaishali Muralidharan, and Ranga Vemuri. “SLED: Sequential Logic Encryption Using Dynamic Keys”. In: *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2020, pp. 844–847. DOI: 10.1109/MWSCAS48704.2020.9184664.

[Kol+19] Gaurav Kolhe et al. “Security and Complexity Analysis of LUT-based Obfuscation: From Blueprint to Reality”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942100.

[Kru] Marcel Krueger. *fontawesome5*. Provided under SIL OFL 1.1 license. https://ctan.org/pkg/fontawesome5.

[KS10] AS Klimovich and VV Solov’ev. “Transformation of a mealy finite-state machine into a moore finite-state machine by splitting internal states”. In: *Journal of Computer and Systems Sciences International* 49 (2010), pp. 900–908. DOI: 10.1134/S1064230710060080. https://doi.org/10.1134/S1064230710060080.

[KTV19] Yasaswy Kasarabada, Sudheer Ram Thulasi Raman, and Ranga Vemuri. “Deep State Encryption for Sequential Logic Circuits”. In: *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2019, pp. 338–343. DOI: 10.1109/ISVLSI.2019.00068.

[Kum00] Jean Kumagai. “Chip detectives [reverse engineering]”. In: *IEEE Spectrum* 37.11 (2000), pp. 43–48.

[KV20] Yasaswy Kasarabada and Ranga Vemuri. “StateLock: State Transition Based Logic Locking for Sequential Circuits”. In: *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*. 2020, pp. 171–176. DOI: 10.1109/VLSID49098.2020.00047.

[LBL21] Matthias Ludwig, Ann-Christin Bette, and Bernhard Lippmann. “ViTaL: Verifying Trojan-Free Physical Layouts through Hardware Reverse Engineering”. In: *2021 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 2021, pp. 1–8. DOI: 10.1109/PAINE54418.2021.9707702.

[Lee22] Hye Hyun Lee. *Misleading Finite State Machines against Sequential Reverse Engineering*. Master Thesis, November 2022. 2022.

[Li+13] Wenchao Li et al. “WordRev: Finding word-level structures in a sea of bit-level gates”. In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. June 2013, pp. 67–74. DOI: 10.1109/HST.2013.6581568.

[Li+18] Meng Li, Kaveh Shamsi, Yier Jin, and David Z. Pan. “TimingSAT: Decamouflaging Timing-based Logic Obfuscation”. In: *2018 IEEE International Test Conference (ITC)*. 2018, pp. 1–10. DOI: 10.1109/TEST.2018.8624671.

[Lim+19] Nimisha Limaye, Abhrajit Sengupta, Mohammed Nabeel, and Ozgur Sinanoglu. “Is Robust Design-for-Security Robust Enough? Attack on Locked Circuits with Restricted Scan Chain Access”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942047.

[Lip+20] Bernhard Lippmann et al. “Verification of physical designs using an integrated reverse engineering flow for nanoscale technologies”. In: *Integration* 71 (2020), pp. 11–29. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.11.005. https://www.sciencedirect.com/science/article/pii/S0167926019302998.

[Lip+23] Bernhard Lippmann et al. “VE-FIDES: Designing Trustworthy Supply Chains Using Innovative Fingerprinting Implementations”. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137026.

[LNO21] Leon Li, Shuyi Ni, and Alex Orailoglu. “JANUS: Boosting Logic Obfuscation Scope Through Reconfigurable FSM Synthesis”. In: *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2021, pp. 292–303. DOI: 10.1109/HOST49136.2021.9702288.

[LO22] Leon Li and Alex Orailoglu. “JANUS-HD: Exploiting FSM Sequentiality and Synthesis Flexibility in Logic Obfuscation to Thwart SAT Attack While Offering Strong Corruption”. In: *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2022, pp. 1323–1328. DOI: 10.23919/DATE54114.2022.9774729.

[LO23] Leon Li and Alex Orailoglu. “Thwarting Reverse Engineering Attacks through Keyless Logic Obfuscation”. In: *2023 IEEE 41st VLSI Test Symposium (VTS)*. 2023, pp. 1–6. DOI: 10.1109/VTS56346.2023.10139952.

[LT15] Yu-Wei Lee and Nur A. Touba. “Improving logic obfuscation via logic cone analysis”. In: *2015 16th Latin-American Test Symposium (LATS)*. 2015, pp. 1–6. DOI: 10.1109/LATW.2015.7102410.

[Lud+21] Matthias Ludwig, Alexander Hepp, Michaela Brunner, and Johanna Baehr. “CRESS: Framework for Vulnerability Assessment of Attack Scenarios in Hardware Reverse Engineering”. In: *2021 IEEE Physical Assurance and Inspection of Electronics (PAINE)*. 2021, pp. 1–8. DOI: 10.1109/PAINE54418.2021.9707695.

[LV62] R. E. Lyons and W. Vanderkulk. “The Use of Triple-Modular Redundancy to Improve Computer Reliability”. In: *IBM Journal of Research and Development* 6.2 (1962), pp. 200–209. DOI: 10.1147/rd.62.0200.

[LWS12] Wenchao Li, Zach Wasson, and Sanjit A. Seshia. “Reverse engineering circuits using behavioral pattern mining”. In: *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. 2012, pp. 83–88. DOI: 10.1109/HST.2012.6224325.

[LZ13] Li Li and Hai Zhou. “Structural transformation for best-possible obfuscation of sequential circuits”. In: *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2013, pp. 55–60. DOI: 10.1109/HST.2013.6581566.

[Mar+18] Hadi Mardani Kamali, Kimia Zamiri Azar, Kris Gaj, Houman Homayoun, and Avesta Sasan. “LUT-Lock: A Novel LUT-Based Logic Obfuscation for FPGA-Bitstream and ASIC-Hardware Protection”. In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2018, pp. 405–410. DOI: 10.1109/ISVLSI.2018.00080.

[Mar+20] Hadi Mardani Kamali, Kimia Zamiri Azar, Houman Homayoun, and Avesta Sasan. “SCRAMBLE: The State, Connectivity and Routing Augmentation Model for Building Logic Encryption”. In: *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 2020, pp. 153–159. DOI: 10.1109/ISVLSI49217.2020.00037.

[McE01] Kenneth S McElvain. *Methods and apparatuses for automatic extraction of finite state machines*. US Patent No. US 6,182,268 B1, Filed Jan. 5th., 1998, Issued Jan. 30th., 2001. Jan. 2001.

[Mea+16] Travis Meade, Yier Jin, Mark Tehranipoor, and Shaojie Zhang. “Gate-level netlist reverse engineering for hardware security: Control logic register identification”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 1334–1337. DOI: 10.1109/ISCAS.2016.7527495.

[Mea+17] Travis Meade, Zheng Zhao, Shaojie Zhang, David Pan, and Yier Jin. “Revisit sequential logic obfuscation: Attacks and defenses”. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017, pp. 1–4. DOI: 10.1109/ISCAS.2017.8050606.

[Mea+18] Travis Meade, Kaveh Shamsi, Thao Le, Jia Di, Shaojie Zhang, and Yier Jin. “The Old Frontier of Reverse Engineering: Netlist Partitioning”. In: *Journal of Hardware and Systems Security* 2.3 (2018), pp. 201–213. ISSN: 2509-3436. DOI: 10.1007/s41635-018-0043-4. https://doi.org/10.1007/s41635-018-0043-4.

[Mea+19] Travis Meade, Jason Portillo, Shaojie Zhang, and Yier Jin. “NETA: When IP Fails, Secrets Leak”. In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ASPDAC ’19. Tokyo, Japan: Association for Computing Machinery, 2019, pp. 90–95. ISBN: 9781450360074. DOI: 10.1145/3287624.3288739. https://doi.org/10.1145/3287624.3288739.

[Mea55] George H. Mealy. “A method for synthesizing sequential circuits”. In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079. DOI: 10.1002/j.1538-7305.1955.tb03788.x.

[Mil19] Michael Mildner. *Fault-Aided Flip-Flop Classification for Reverse Engineering of Block Cipher Implementations*. Master Thesis, December 2019. 2019.

[MLS23] Priya Mittu, Yuntao Liu, and Ankur Srivastava. “TimingCamouflage+ Decamouflaged”. In: *Proceedings of the Great Lakes Symposium on VLSI 2023*. GLSVLSI ’23. Knoxville, TN, USA: Association for Computing Machinery, 2023, pp. 575–580. ISBN: 9798400701252. DOI: 10.1145/3583781.3590238. https://doi.org/10.1145/3583781.3590238.

[MO98] J.C. Monteiro and A.L. Oliveria. “Finite state machine decomposition for low power”. In: *Proceedings 1998 Design and Automation Conference. 35th DAC. (Cat. No.98CH36175)*. 1998, pp. 758–763. DOI: 10.1145/277044.277235.

[Moh+21] Prashanth Mohan, Oguz Atli, Joseph Sweeney, Onur Kibar, Larry Pileggi, and Ken Mai. “Hardware Redaction via Designer-Directed Fine-Grained eFPGA Insertion”. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2021, pp. 1186–1191. DOI: 10.23919/DATE51398.2021.9473910.

[Moo56] Edward F. Moore. “Gedanken-experiments on sequential machines”. In: *Automata studies* 34 (1956), pp. 129–153.

[MS21] Ivan Miketic and Emre Salman. “PhaseCamouflage: Leveraging Adiabatic Operation to Thwart Reverse Engineering”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 29.7 (2021), pp. 1285–1296. DOI: 10.1109/TVLSI.2021.3078567.

[MZJ16] Travis Meade, Shaojie Zhang, and Yier Jin. “Netlist reverse engineering for high-level functionality reconstruction”. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 655–660. DOI: 10.1109/ASPDAC.2016.7428086.

[MZJ17] Travis Meade, Shaojie Zhang, and Yier Jin. “IP protection through gate-level netlist security enhancement”. In: *Integration* 58 (2017), pp. 563–570. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2016.10.014. https://www.sciencedirect.com/science/article/pii/S0167926016300827.

[MZJ19] Travis Meade, Shaojie Zhang, and Yier Jin. *NetA*. Oct. 2019. https://github.com/jinyier/NetA.

[Nah+16] Adib Nahiyan, Kan Xiao, Kun Yang, Yeir Jin, Domenic Forte, and Mark Tehranipoor. “AVFSM: A Framework for Identifying and Mitigating Vulnerabilities in FSMs”. In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC ’16. Austin, Texas: Association for Computing Machinery, 2016. ISBN: 9781450342360. DOI: 10.1145/2897937.2897992. https://doi.org/10.1145/2897937.2897992.

[Okl15] Oklahoma State University. *Flows/FreePDK45*. 2015. https://vlsiarch.ecen.okstate.edu/flow/.

[Onc17a] OnchipUIS. *mriscv*. Available at https://github.com/onchipuis/mriscv. 2017.

[Onc17b] OnchipUIS. *mriscvcore*. Available at https://github.com/onchipuis/mriscvcore. 2017.

[Ope] OpenCores. *OpenCores*. https://opencores.org/projects.

[Ope10] OpenCores. *Galois Counter Mode Advanced Encryption Standard GCM-AES*. Available at https://opencores.org/projects/gcm-aes. 2010.

[Ope15] OpenCores. *AltOr32 - Alternative Lightweight OpenRisc CPU*. Available at https://opencores.org/projects/altor32. 2015.

[Ope16a] OpenCores. *AES128*. Available at https://opencores.org/projects/apbtoaes128. 2016.

[Ope16b] OpenCores. *Documented Verilog UART*. Available at https://opencores.org/projects/osdvu. 2016.

[Ope18] OpenCores. *FT816Float - Floating point accelerator*. Available at https://opencores.org/projects/ft816float. 2018.

[Ope19] Opencircuitdesign. *qflow*. 2019. http://opencircuitdesign.com/qflow/.

[Pat+18] Ahmad Patooghy, Ehsan Aerabi, Hamidreza Rezaei, Miguel Mark, Mahdi Fazeli, and Michel A Kinsy. “Mystic: Mystifying IP cores using an always-ON FSM obfuscation method”. In: *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE. 2018, pp. 626–631. DOI: 10.1109/ISVLSI.2018.00119.

[PCB07] Somnath Paul, Rajat Subhra Chakraborty, and Swarup Bhunia. “VIm-Scan: A Low Overhead Scan Design Approach for Protection of Secret Key in Scan-Based Secure Chips”. In: *25th IEEE VLSI Test Symposium (VTS’07)*. 2007, pp. 455–460. DOI: 10.1109/VTS.2007.89.

[Ped+11] Fabian Pedregosa et al. “Scikit-Learn: Machine Learning in Python”. In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435.

[Ped13] Volnei A. Pedroni. *Finite state machines in hardware: theory and design (with VHDL and SystemVerilog)*. The MIT Press, 2013. ISBN: 9780262319096. DOI: 10.7551/mitpress/9657.001.0001. https://doi.org/10.7551/mitpress/9657.001.0001.

[PM15] Stephen M. Plaza and Igor L. Markov. “Solving the Third-Shift Problem in IC Piracy With Test-Aware Logic Locking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.6 (2015), pp. 961–971. DOI: 10.1109/TCAD.2015.2404876.

[Put19] Maximilian Putz. *Analysis of the NETA Toolset and Comparison with a New Optimized fastRELIC Implementation*. Bachelor Thesis, July 2019. 2019.

[Qua+16] Shahed E. Quadir et al. “A Survey on Chip to System Reverse Engineering”. In: *J. Emerg. Technol. Comput. Syst.* 13.1 (Apr. 2016), pp. 1–34. ISSN: 1550-4832. DOI: 10.1145/2755563. https://doi.org/10.1145/2755563.

[Rah+20a] M Tanjidur Rahman, Shahin Tajik, M Sazadur Rahman, Mark Tehranipoor, and Navid Asadizanjani. “The Key is Left under the Mat: On the Inappropriate Security Assumption of Logic Locking Schemes”. In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 262–272. DOI: 10.1109/HOST45689.2020.9300258.

[Rah+20b] M. Tanjidur Rahman et al. “Defense-in-depth: A recipe for logic locking to prevail”. In: *Integration* 72 (2020), pp. 39–57. ISSN: 0167-9260. DOI: https://doi.org/10.1016/j.vlsi.2019.12.007. https://www.sciencedirect.com/science/article/pii/S0167926019303694.

[Rah+23] M. Sazadur Rahman, Rui Guo, Hadi M. Kamali, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. “ReTrustFSM: Toward RTL Hardware Obfuscation-A Hybrid FSM Approach”. In: *IEEE Access* 11 (2023), pp. 19741–19761. DOI: 10.1109/ACCESS.2023.3244902.

[Raj+12a] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. “Logic encryption: A fault analysis perspective”. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2012, pp. 953–958. DOI: 10.1109/DATE.2012.6176634.

[Raj+12b] Jeyavijayan Rajendran, Youngok Pino, Ozgur Sinanoglu, and Ramesh Karri. “Security analysis of logic obfuscation”. In: *Proceedings of the 49th Annual Design Automation Conference*. ACM. 2012, pp. 83–89. ISBN: 9781450311991. DOI: 10.1145/2228360.2228377. https://doi.org/10.1145/2228360.2228377.

[Raj+15] Jeyavijayan Rajendran et al. “Fault Analysis-Based Logic Encryption”. In: *IEEE Transactions on Computers* 64.2 (2015), pp. 410–424. DOI: 10.1109/TC.2013.193.

[RB22] Md Moshiur Rahman and Swarup Bhunia. “Practical Implementation of Robust State Space Obfuscation for Hardware IP Protection”. In: (Nov. 2022). DOI: 10.36227/techrxiv.21405075.v1. https://www.techrxiv.org/articles/preprint/Practical_Implementation_of_Robust_State_Space_Obfuscation_for_Hardware_IP_Protection/21405075.

[RKM08a] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. “EPIC: Ending Piracy of Integrated Circuits”. In: *2008 Design, Automation and Test in Europe*. 2008, pp. 1069–1074. DOI: 10.1109/DATE.2008.4484823.

[RKM08b] Jarrod A. Roy, Farinaz Koushanfar, and Igor L. Markov. “Protecting bus-based hardware IP by secret sharing”. In: *2008 45th ACM/IEEE Design Automation Conference*. 2008, pp. 846–851. DOI: 10.1145/1391469.1391684.

[RMS18] Shervin Roshanisefat, Hadi Mardani Kamali, and Avesta Sasan. “SRCLock: SAT-Resistant Cyclic Logic Locking for Protecting the Hardware”. In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. GLSVLSI ’18. Chicago, IL, USA: Association for Computing Machinery, 2018, pp. 153–158. ISBN: 9781450357241. DOI: 10.1145/3194554.3194596. https://doi.org/10.1145/3194554.3194596.

[Ros+20] Shervin Roshanisefat et al. “DFSSD: Deep Faults and Shallow State Duality, A Provably Strong Obfuscation Solution for Circuits with Restricted Access to Scan Chain”. In: *2020 IEEE 38th VLSI Test Symposium (VTS)*. 2020, pp. 1–6. DOI: 10.1109/VTS48691.2020.9107629.

[Ros+21] Shervin Roshanisefat, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. “RANE: An Open-Source Formal De-Obfuscation Attack for Reverse Engineering of Logic Encrypted Circuits”. In: *Proceedings of the 2021 on Great Lakes Symposium on VLSI (GLSVLSI)*. ACM, 2021, pp. 221–228. ISBN: 9781450383936. DOI: 10.1145/3453688.3461760.

[Sah+21] Akashdeep Saha, Hrivu Banerjee, Rajat Subhra Chakraborty, and Debdeep Mukhopadhyay. “ORACALL: An Oracle-Based Attack on Cellular Automata Guided Logic Locking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.12 (2021), pp. 2445–2454. DOI: 10.1109/TCAD.2021.3050035.

[Sha+17] Kaveh Shamsi, Meng Li, Travis Meade, Zheng Zhao, David Z. Pan, and Yier Jin. “Cyclic Obfuscation for Creating SAT-Unresolvable Circuits”. In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. GLSVLSI ’17. Banff, Alberta, Canada: Association for Computing Machinery, 2017, pp. 173–178. ISBN: 9781450349727. DOI: 10.1145/3060403.3060458. https://doi.org/10.1145/3060403.3060458.

[Sha+18] Kaveh Shamsi, Meng Li, David Z. Pan, and Yier Jin. “Cross-Lock: Dense Layout-Level Interconnect Locking Using Cross-Bar Architectures”. In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. GLSVLSI ’18. Chicago, IL, USA: Association for Computing Machinery, 2018, pp. 147–152. ISBN: 9781450357241. DOI: 10.1145/3194554.3194580. https://doi.org/10.1145/3194554.3194580.

[Shi+10] Yiqiong Shi, Chan Wai Ting, Bah-Hwee Gwee, and Ye Ren. “A highly efficient method for extracting FSMs from flattened gate-level netlist”. In: *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2010, pp. 2610–2613. DOI: 10.1109/ISCAS.2010.5537093.

[Shi+12] Yiqiong Shi, Bah-Hwee Gwee, Ye Ren, Thet Khaing Phone, and Chan Wai Ting. “Extracting functional modules from flattened gate-level netlist”. In: *2012 International Symposium on Communications and Information Technologies (ISCIT)*. 2012, pp. 538–543. DOI: 10.1109/ISCIT.2012.6380958.

[SIA22] SIA. *Number of new semiconductor fabrication projects worldwide in 2021, by region*. 2022. https://www.statista.com/statistics/1304271/number-semiconductor-fabrication-projects-by-region/.

[SLG21] Aayush Singla, Bernhard Lippmann, and Helmut Graeb. “Recovery of 2D and 3D Layout Information through an Advanced Image Stitching Algorithm using Scanning Electron Microscope Images”. In: *2020 25th International Conference on Pattern Recognition (ICPR)*. 2021, pp. 3860–3867. DOI: 10.1109/ICPR48806.2021.9412334.

[SRM15] Pramod Subramanyan, Sayak Ray, and Sharad Malik. “Evaluating the security of logic encryption algorithms”. In: *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2015, pp. 137–143. DOI: 10.1109/HST.2015.7140252.

[Sto+21] Florian Stolz et al. “LifeLine for FPGA Protection: Obfuscated Cryptography for Real-World Security”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.4 (Aug. 2021), pp. 412–446. DOI: 10.46586/tches.v2021.i4.412-446. https://tches.iacr.org/index.php/TCHES/article/view/9071.

[Str16] Joachim Strömbergson. *secworks siphash*. Available at https://github.com/secworks/siphash/tree/master/src/rtl. 2016.

[Str18a] Joachim Strömbergson. *secworks aes*. Available at https://github.com/secworks/aes/tree/master/src/rtl. 2018.

[Str18b] Joachim Strömbergson. *secworks sha1*. Available at https://github.com/secworks/sha1/tree/master/src/rtl. 2018.

[Sub+13] Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. “Reverse engineering digital circuits using functional analysis”. In: *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2013, pp. 1277–1280. DOI: 10.7873/DATE.2013.264.

[Sub+14] Pramod Subramanyan et al. “Reverse engineering digital circuits using structural and functional analyses”. In: *IEEE Transactions on Emerging Topics in Computing* 2.1 (2014), pp. 63–80. DOI: 10.1109/TETC.2013.2294918.

[Swe+20] Joseph Sweeney, V Mohammed Zackriya, Samuel Pagliarini, and Lawrence Pileggi. “Latch-Based Logic Locking”. In: *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 132–141. DOI: 10.1109/HOST45689.2020.9300256.

[Syk21] Geralda Syku. *Implementation of an Automatic Redesign of State Machines for Obfuscation*. Bachelor Thesis, September 2021. 2021.

[Tar71] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 1971, pp. 114–121. DOI: 10.1109/SWAT.1971.10.

[Thi20] Paul Thiele. *Attacks on Hardware Obfuscation with Machine Learning Based Reverse Engineering*. Master Thesis, January 2020. 2020.

[TJ09] Randy Torrance and Dick James. “The State-of-the-Art in IC Reverse Engineering”. In: *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*. CHES ’09. Lausanne, Switzerland: Springer-Verlag, 2009, pp. 363–381. ISBN: 9783642041372. DOI: 10.1007/978-3-642-04138-9_26. https://doi.org/10.1007/978-3-642-04138-9_26.

[UGP23] Devanshi Upadhyaya, Maël Gay, and Ilia Polian. “LEDA: Locking Enabled Differential Analysis of Cryptographic Circuits”. In: *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2023, pp. 249–259. DOI: 10.1109/HOST55118.2023.10133696.

[Uss02] Rudolf Usselmann. *ASICS/FPGA Design House aes_128*. Available at https://github.com/www-asics-ws/aes_128. 2000-2002.

[Utm21] Utmel. *What is Chip: Definition, Classification and Design Process*. Accessed on March 2023. Dec. 2021. https://www.utmel.com/blog/categories/integrated%20circuit/what-is-chip-definition-classification-and-design-process.

[Wan+11] Xinmu Wang, Seetharam Narasimhan, Aswin Krishna, Tatini Mal-Sarkar, and Swarup Bhunia. “Sequential hardware Trojan: Side-channel aware design and placement”. In: *2011 IEEE 29th International Conference on Computer Design (ICCD)*. 2011, pp. 297–300. DOI: 10.1109/ICCD.2011.6081413.

[Web+22] Selina Weber, Johanna Baehr, Alexander Hepp, and Georg Sigl. “Analysis of Graph-based Partitioning Algorithms and Partitioning Metrics for Hardware Reverse Engineering”. In: *11th International Workshop on Security Proofs for Embedded Systems (PROOFS 2022)*. Leuven, Belgium, Sept. 2022. https://hal.science/hal-03780642.

[Wer+18] Michael Werner, Bernhard Lippmann, Johanna Baehr, and Helmut Gräb. “Reverse Engineering of Cryptographic Cores by Structural Interpretation Through Graph Analysis”. In: *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*. 2018, pp. 13–18. DOI: 10.1109/IVSW.2018.8494896.

[Wie+19] Carina Wiesen et al. “Towards Cognitive Obfuscation: Impeding Hardware Reverse Engineering Based on Psychological Insights”. In: *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ASPDAC ’19. Tokyo, Japan: Association for Computing Machinery, 2019, pp. 104–111. ISBN: 9781450360074. DOI: 10.1145/3287624.3288741. https://doi.org/10.1145/3287624.3288741.

[Wol18] Clifford Wolf. *Yosys Open SYnthesis Suite*. 2018. https://yosyshq.net/yosys/.

[WP13] Sheng Wei and Miodrag Potkonjak. “The undetectable and unprovable hardware Trojan horse”. In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2013, pp. 1–2. DOI: 10.1145/2463209.2488912.

[XS16] Yang Xie and Ankur Srivastava. “Mitigating SAT Attack on Logic Locking”. In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 127–146. ISBN: 978-3-662-53140-2.

[XS17] Yang Xie and Ankur Srivastava. “Delay locking: Security enhancement of logic locking against IC counterfeiting and overproduction”. In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2017, pp. 1–6. DOI: 10.1145/3061639.3062226.

[Xu+17] Xiaolin Xu, Bicky Shakya, Mark M. Tehranipoor, and Domenic Forte. “Novel Bypass Attack and BDD-based Tradeoff Analysis Against All Known Logic Locking Attacks”. In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Cham: Springer International Publishing, 2017, pp. 189–210. ISBN: 978-3-319-66787-4.

[Xu+23] Hongye Xu, Dongfang Liu, Cory Merkel, and Michael Zuzak. “Exploiting Logic Locking for a Neural Trojan Attack on Machine Learning Accelerators”. In: GLSVLSI ’23 (2023), pp. 351–356. DOI: 10.1145/3583781.3590242. https://doi.org/10.1145/3583781.3590242.

[Yas+16a] Muhammad Yasin, Bodhisatwa Mazumdar, Jeyavijayan J V Rajendran, and Ozgur Sinanoglu. “SARLock: SAT attack resistant logic locking”. In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2016, pp. 236–241. DOI: 10.1109/HST.2016.7495588.

[Yas+16b] Muhammad Yasin, Jeyavijayan JV Rajendran, Ozgur Sinanoglu, and Ramesh Karri. “On Improving the Security of Logic Locking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.9 (2016), pp. 1411–1424. DOI: 10.1109/TCAD.2015.2511144.

[Yas+17a] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. “Security analysis of Anti-SAT”. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, pp. 342–347. DOI: 10.1109/ASPDAC.2017.7858346.

[Yas+17b] Muhammad Yasin, Abhrajit Sengupta, Mohammed Thari Nabeel, Mohammed Ashraf, Jeyavijayan (JV) Rajendran, and Ozgur Sinanoglu. “Provably-Secure Logic Locking: From Theory To Practice”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1601–1618. ISBN: 9781450349468. DOI: 10.1145/3133956.3133985. https://doi.org/10.1145/3133956.3133985.

[Yas+17c] Muhammad Yasin, Abhrajit Sengupta, Benjamin Carrion Schafer, Yiorgos Makris, Ozgur Sinanoglu, and Jeyavijayan (JV) Rajendran. “What to Lock? Functional and Parametric Locking”. In: *Proceedings of the on Great Lakes Symposium on VLSI 2017*. GLSVLSI ’17. Banff, Alberta, Canada: Association for Computing Machinery, 2017, pp. 351–356. ISBN: 9781450349727. DOI: 10.1145/3060403.3060492. https://doi.org/10.1145/3060403.3060492.

[Yas+20] Muhammad Yasin, Bodhisatwa Mazumdar, Ozgur Sinanoglu, and Jeyavijayan Rajendran. “Removal Attacks on Logic Locking and Camouflaging Techniques”. In: *IEEE Transactions on Emerging Topics in Computing* 8.2 (2020), pp. 517–532. DOI: 10.1109/TETC.2017.2740364.

[YC16] Cunxi Yu and Maciej Ciesielski. “Automatic word-level abstraction of datapath”. In: *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2016, pp. 1718–1721. DOI: 10.1109/ISCAS.2016.7538899.

[Yos22] YosysHQ. *picorv32*. Available at https://github.com/YosysHQ/picorv32. 2022.

[Zha+18a] Fan Zhang et al. “Persistent Fault Analysis on Block Ciphers”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (Aug. 2018), pp. 150–172. DOI: 10.13154/tches.v2018.i3.150-172. https://tches.iacr.org/index.php/TCHES/article/view/7272.

[Zha+18b] Grace Li Zhang, Bing Li, Bei Yu, David Z. Pan, and Ulf Schlichtmann. “TimingCamouflage: Improving circuit security against counterfeiting by unconventional timing”. In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018, pp. 91–96. DOI: 10.23919/DATE.2018.8341985.

[Zha+19] Tao Zhang, Jian Wang, Shize Guo, and Zhe Chen. “A Comprehensive FPGA Reverse Engineering Tool-Chain: From Bitstream to RTL Code”. In: *IEEE Access* 7 (2019), pp. 38379–38389. DOI: 10.1109/ACCESS.2019.2901949.

[Zha+20a] Grace Li Zhang, Michaela Brunner, Bing Li, Georg Sigl, and Ulf Schlichtmann. “Timing Resilience for Efficient and Secure Circuits”. In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2020, pp. 623–628. DOI: 10.1109/ASP-DAC47756.2020.9045352.

[Zha+20b] Grace Li Zhang et al. “TimingCamouflage+: Netlist Security Enhancement With Unconventional Timing”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.12 (2020), pp. 4482–4495. DOI: 10.1109/TCAD.2020.2974338.

[ZSR19] Hai Zhou, Yuanqi Shen, and Amin Rezaei. *Vulnerability and Remedy of Stripped Function Logic Locking*. Cryptology ePrint Archive, Paper 2019/139. 2019. https://eprint.iacr.org/2019/139.

Credits: