

Doppelganger Obfuscation — Exploring the Defensive and Offensive Aspects of Hardware Camouflaging

Max Hoffmann^{1,2} and Christof Paar^{1,2}

¹ Horst Görtz Institute for IT-Security, Ruhr University Bochum, Germany

² Max Planck Institute for Security and Privacy, Germany

max.hoffmann@rub.de, christof.paar@rub.de

Abstract. Hardware obfuscation is widely used in practice to counteract reverse engineering. In recent years, low-level obfuscation via *camouflaged gates* has been increasingly discussed in the scientific community and industry. In contrast to classical high-level obfuscation, such gates result in recovery of an erroneous netlist. This technology has so far been regarded as a purely defensive tool. We show that low-level obfuscation is in fact a double-edged sword that can also enable stealthy malicious functionalities.

In this work, we present Doppelganger, the first generic design-level obfuscation technique that is based on low-level camouflaging. Doppelganger obstructs central control modules of digital designs, e.g., Finite State Machines (FSMs) or bus controllers, resulting in two different design functionalities: an apparent one that is recovered during reverse engineering and the actual one that is executed during operation. Notably, both functionalities are under the designer’s control.

In two case studies, we apply Doppelganger to a universal cryptographic coprocessor. First, we show the defensive capabilities by presenting the reverse engineer with a different mode of operation than the one that is actually executed at an area overhead of less than 0.6%. Then, for the first time, we demonstrate the considerable threat potential through low-level obfuscation. We show how an invisible, remotely exploitable key-leakage Trojan can be injected into the same cryptographic coprocessor just through obfuscation at an area overhead of mere 0.01%.

Keywords: Hardware Obfuscation · Gate-level · Camouflaging · Hardware Trojans

1 Introduction

Hardware-based threats have become increasingly important in recent years and have moved into the public discussion, for example, in the 2018-Bloomberg allegations about a supposed hardware backdoor in Supermicro server hardware [RR18], or the recent ban by the US government on telecommunication equipment from China [KS]. In the work at hand, we focus on an important aspect of hardware security, namely protection against hardware reverse engineering. Such protection is crucial in several scenarios, including (1) protection of Intellectual Property (IP), (2) impeding the insertion of hardware Trojans by third parties, and (3) hiding security- or safety-critical components. First, *IP-theft* is of great concern for virtually all industries, ranging from consumer electronics to the large-scale cyber-physical systems. The Semiconductor Equipment and Materials International Association (SEMI) reported losses from IP-theft between \$2 billion and \$4 billion already in 2008 [Sem08]. Second, preventing malicious *hardware manipulations* is another important effect of defending against hardware reverse engineering. The fear of such manipulations is also a backdrop for the current discussions on the possibility of

hidden backdoors in foreign-built computer and communication equipment [KS]. Finally, hiding critical components, e.g., in TPMs, HSMs, smart cards, and larger designs that contain security modules, is a widely used defense strategy to reduce the surface for attack vectors such as targeted key extraction, laser-based fault attacks, or weakening of TRNGs.

Apart from physical protection of Integrated Circuits (ICs), e.g., via active shields or coatings, the primary method to defend the design itself against reverse engineering is through *obfuscation*. By definition, an obfuscation is a transformation that maintains functional equivalence but obstructs comprehensibility. Hence, obfuscation aims to increase reversing cost to extents where it becomes infeasible for an attacker. We would like to stress that hardware obfuscation is not a niche technology but widely used in the design industry, e.g., commercial IP cores are routinely obfuscated.

Hardware obfuscation methods can be divided in two families: traditional high-level and hidden low-level obfuscation. Traditional high-level obfuscation is applied in order to increase the difficulty of *understanding the netlist* of an IC, examples include bus scrambling, randomized routing, and dummy states in FSMs. Note that an adversary has all information about the design under attack and given enough effort, will be successful. More recently, low-level obfuscation has been increasingly discussed in academia. In stark contrast to traditional obfuscation, its goal is to create a situation in which the reverse engineer can only recover an incomplete or *erroneous netlist* through almost invisible alterations to logic cells or even single transistors. For instance, by changing the dopant concentration of transistors, a gate can have a different functionality from the apparent one recovered during reverse engineering. We note that low-level obfuscation (in contrast to traditional obfuscation) creates an asymmetry between the designer and the analyst, because the latter recovers incomplete or even incorrect information about the design under attack. Interestingly, while such low-level techniques have been addressed over the past seven years in academia [RSSK13, EEAM16, CEMG16, MBPB15, LSM⁺17, PAKS17, SSTF19], patents [BCCJ98, CBCJ07, CBW⁺12, CBWC12], and even by companies that offer camouflaged gates as a service [Ram], they were purely presented as atomic building blocks. Little is known how to leverage these building blocks in novel high-level techniques for generic obfuscation of (large) designs.

In this work, we close this gap by presenting Doppelganger, a generic and extremely stealthy hardware obfuscation technique. Doppelganger is applicable to arbitrary encoding logic, for instance, FSM state transitions or address resolution. Especially the former is an attractive target for obfuscation since in more complex designs, the control logic often contains the majority of the high-level information a reverse engineer may want to discover. At its core, Doppelganger-generated logic presents a different visible behavior to a reverse engineer than the one which is actually executed. On the physical level, our technique makes use of camouflaged gates with dummy inputs and can be instantiated independently of the physical realization of said primitive. We demonstrate that Doppelganger comes at almost negligible overhead and offers the designer control over the visible outcome of the obfuscation, which enables much more stealthy results than traditional approaches.

As mentioned above, obfuscation is traditionally regarded as a purely defensive countermeasure. However, obfuscation is in fact a double-edged sword. In this work we, for the first time, highlight alarming implications of offensive use of low-level obfuscation. Using Doppelganger, we trojanize a cryptographic coprocessor in a way that raises no suspicion, even upon detailed analysis, yet enables remote exploitation.

Contribution: Our three main contributions are:

- We present Doppelganger, a novel hardware obfuscation technique to generate logic which presents a different behavior to a reverse engineer than the one which is actually implemented. It is generically applicable to arbitrary encoding or addressing logic as found in virtually any digital design. Notably, our technique infers minuscule overhead, is (conveniently) applied at the design stage in human-readable format,

and gives the designer control over the obfuscated version of the design, which the reverse engineer will face.

- We demonstrate the **defensive** strength and applicability of Doppelganger in a case study where we obstruct the central control logic of a cryptographic coprocessor. Our results demonstrate the power of our method by generating an apparent functionality that is still plausible. We further show that, depending on the application, even a randomized apparent functionality can result in strong obfuscation.
- To highlight the **offensive** implications of low-level obfuscation, we take the point-of-view of a malicious designer. We show that Doppelganger can be used to inject an extremely stealthy hardware Trojan into the same cryptographic coprocessor. Our exemplary Trojan — coined Evil Twin — is the first demonstration of a kleptographic Algorithm Substitution Attack (ASA) in hardware and allows for remote exploitation. We argue that our Trojan remains undetected when faced with the majority of conventional Trojan detection approaches.

Availability: All algorithms, hardware implementations, and evaluations will be made available open-source via GitHub.

2 Technical Background

In this section, we will provide background knowledge for selected parts of our contribution. We will discuss hardware reverse engineering, low-level obfuscation, and briefly survey related work. We recall the definition of FSMs and explain how they are realized in hardware, as this is necessary for our case studies.

2.1 Hardware Reverse Engineering

Hardware reverse engineering can be split into two phases: netlist recovery and netlist analysis. Netlist recovery from an IC consists of four time-consuming steps [QCF⁺16, CC90, TJ11], namely (1) decapsulation, (2) delayering, (3) imaging and (4) post-processing. During decapsulation, the die is extracted from its surrounding package. The die is then delayered via a mixture of chemical and mechanical etching, while several high-resolution images of the surface of each layer are taken. The standard procedure for image acquisition in advanced labs is combining optical imaging for general structures with Scanning Electron Microscope (SEM) imaging for small features [TJ11, BWL⁺20] which results in several hundred to thousand images per layer. During post-processing, they are aligned, annotated, and standard cells and their interconnections across multiple layers are identified. These steps result in the netlist, an abstracted representation of the design, equivalent to a large circuit diagram. Note that this phase is *destructive*, hence if errors occur that cannot be corrected retrospectively, the process has to be (partially) repeated with a new IC.

The second phase is netlist analysis, which heavily depends on the analyst’s objective. However, it is typically a mixture of manual and semi-automated analyses to gain an *understanding* of critical parts. An example is the semi-automated reverse engineering of high level modules, which is a common initial step, cf. [S⁺12, L⁺13, M⁺16b, M⁺16a, FWS⁺18]. If the netlist that was recovered in first phase is faulty, e.g., because of low-level obfuscation, gaining a correct understanding of the analyzed device can be considerably impeded.

2.2 Low-Level Obfuscation

Obfuscation is considered as the primary countermeasure against reverse engineering [MBPB15, VPH⁺17]. The main difference between traditional high-level and low-level

hardware obfuscation (cf. Section 1) is that the former complicates analysis of a recovered netlist, while the latter increases the difficulty of recovering the netlist itself, as it results in an erroneous netlist. Crucially, an adversary that is faced with a traditional obfuscation is working with a correct netlist, while he has to cope with an unknown level of uncertainty when faced with low-level obfuscation. In the latter case, one or more gates possess a functionality different from what is apparent (camouflaging) or seemingly connected wires are actually floating (dummy wires). In contrast to traditional obfuscation, low-level obfuscation is particularly vulnerable to dynamic analysis, if there is an observable mismatch between expected and obtained outputs: the more precisely an analyst can trace the impact of single gates on the output, the easier he can locate and deobfuscate camouflaged cells. However, this isolated analysis is often not possible in practice due to the many levels of registers and logic between primary inputs and outputs.

Particularly relevant for the work at hand is gate- or transistor-level obfuscation, i.e., camouflaging. In the following we briefly present related work in this area.

Related Work: Previous work has almost exclusively focused on providing camouflaged circuit elements as atomic building blocks. This includes several Google patents [BCCJ98, CBCJ07, CBW⁺12, CBWC12] and multiple academic publications: Rajendran et al. presented a camouflaged gate that can implement either an OR, AND, or an XOR gate [RSSK13], i.e, a one-of-many gate. Their construction features a comparatively large number of transistors connected through a matrix of wires, where only a few connections are actually conducting while others are separated via small isolation layers. Therefore, their camouflaged gates can be easily identified on the layer images, but the implemented functionality of the gate is not deducible from those. In parallel, Erbagci et al. and Colantes et al. proposed constructions for one-of-many gates based on the transistor threshold voltage asserted during manufacturing [EEAM16, CEMG16]. Malik et al. proposed a standard cell library that integrates gate-level obfuscation through dopant manipulations into standard design tools [MBPB15]. While dopant changes remain invisible during the standard reverse engineering process they can be recovered with notable additional effort per gate [SSF⁺14]. Li et al. formally analyzed resistance of gate camouflaging against SAT-solver-based reverse engineering, revisited dopant-based modification to facilitate always-on and always-off transistors, and demonstrated how to securely obfuscate AND-trees [LSM⁺17]. In [PAKS17], Patnaik et al. focused on obfuscating the vias that interconnect gates and wires. Their approach basically generates several multi-driven gate inputs where only one driver is actually connected. Recently, Shakya et al. revisited dopant-based camouflaging in [SSTF19]. They describe how shallow manipulations in the channels of transistors can be used to effectively turn them into always-on or always-off transistors without being detected as shown by [SSF⁺14]. Using these building blocks they present a generic technique to extend a combinational gate by an arbitrary number of dummy inputs. Crucially, they demonstrate indistinguishability of their always-on/-off transistors from genuine transistors using modern imaging techniques on a real chip. Low-level obfuscation is also already commercially available, e.g., in the SypherMedia Library (SML) Circuit Camouflage Technology by Rambus [Ram].

Perhaps somewhat surprisingly, there are no reports in the open literature on how to employ camouflaged gates to obfuscate more complex high-level structures, which is the objective of our contribution.

2.3 Finite State Machines (FSMs)

Hardware designs are typically separated in a data path and a control path. While the data path carries out computations and features interconnecting buses, the control path orchestrates all modules in the design and handles the control signals of the data path. Said control logic is typically implemented via a cascade of Finite State Machines (FSMs).

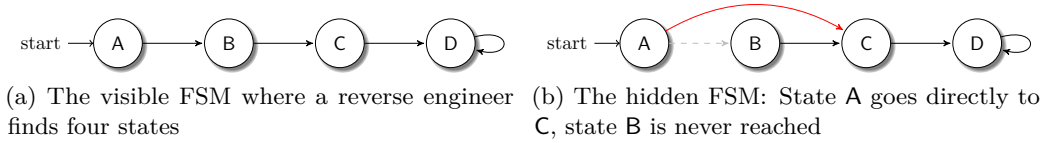


Figure 1: A simple Doppelganger FSM

While Doppelganger is a generic technique for addressing and encoding logic, in our case studies we use it primarily to obfuscate the state transitions of an FSM. Since reversing FSMs is typically crucial for understanding high-level functionalities, obfuscating them is a promising strategy.

Hardware FSMs consist of three modules, namely (1) a state register, (2) feedback logic, and (3) output logic [FWD⁺18]. The state register holds the encoded current state, typical encodings are one-hot, binary, or Gray code. The feedback logic takes the current state together with additional signals from the remaining circuitry and computes the next state based on these signals. The output logic computes control signals for the data path or child-FSMs from the bits of the state register.

3 Obfuscation with Doppelganger

In this section we present Doppelganger, a novel obfuscation technique that is applicable to addressing logic or encoding logic. Especially attractive targets for Doppelganger obfuscation are FSM state transition functions (encoding logic), since they are the central control elements in digital designs, or bus address resolution modules (addressing logic), since buses control the data flow of the design. For clarity, we will collectively refer to encoding and addressing as *encoding* in the following.

We start with a high-level overview on Doppelganger, followed by a short example. Then, we provide a detailed algorithmic descriptions. The strength and stealthiness of Doppelganger are evaluated in Sections 4, 5, and 6 and limitations are discussed in Section 7.

Overview: The overarching goal of Doppelganger is to generate a design that, when reverse engineered, yields a netlist that does not match the actual functionality of the design. While a reverse engineer will recover a seemingly valid netlist that appears to implement functionality *A*, the IC actually executes a different functionality *B*. In the following we will refer to *A* as the *visible functionality* and to *B* as the *hidden functionality*. As a unique feature, the designer has control over **both** functionalities. This is in contrast to many other obfuscation schemes, where the designer cannot control the output that is visible to a reverse engineer.

In a nutshell, Doppelganger achieves this dual-functionality by *ignoring* parts of the combinational logic via camouflaged gates. Applied during the conceptual phase of the design process, i.e., on HDL level, Doppelganger takes two high-level descriptions as input: one of the visible functionality and one of the hidden functionality. Its output consists of two components: The first one is a synthesizable design that implements the visible functionality. The second component is a set of specific connections between nets and cells. When the specified connections are ignored in the computations by replacing the respective gates with their camouflaged counterparts, the design now executes the hidden functionality. Since a reverse engineer cannot tell the camouflaged gates apart from benign gates, he is not aware that parts of the circuitry are effectively ignored, and therefore still recovers a netlist of the visible functionality.

Adversarial Model: The “adversary” is a reverse engineer, facing an unknown IC that features hidden functionality generated with Doppelganger. His goal is to recover (parts

Table 1: Exemplary encodings of the states for our FSM

State	A	B	C	D
Encoding $[s_1:s_0]$	00	11	01	10

of) the actual functionality, i.e., the hidden functionality, of the IC. The reverse engineer has access to several fabricated ICs. Note, that the adversary is an actor that operates after fabrication, i.e., Doppelganger is not suitable to defend, e.g., against a malicious foundry.

Requirements: In order to successfully apply Doppelganger at full strength, the designer has to be able to instantiate camouflaged gates with dummy inputs in the fabrication process, e.g., as offered commercially by Rambus [Ram]. Furthermore, these camouflaged gates must be indistinguishable from their benign counterparts during netlist recovery given the standard reversing procedure, as demonstrated in [SSTF19]. Note that, while dedicated deobfuscation methods may exist, they tend to infer high costs [SSF⁺14] and the large design space for camouflaged gates in general [VPH⁺17] makes it impossible to apply them to substantial parts of an ICs during a standard reversing process, even for moderately-sized designs.

3.1 An Example of Doppelganger

In this section, we will illustrate the workflow of Doppelganger by means of a simple example. As mentioned above, Doppelganger is applied to obfuscate encoding logic and an especially attractive target are hence FSMs, which are present in virtually all digital designs. Doppelganger can be applied to the feedback logic which determines the state transitions, i.e., the *next state* signal.

We consider the FSMs shown in Figure 1. The designer wants the reverse engineer to recover the visible FSM from Figure 1a, while the actually executed functionality is the hidden FSM shown in Figure 1b. The first step is to select suitable encodings for the states. Assume the encodings of the 2-bit FSM state register \mathbf{s} shown in Table 1. These encodings result in the following Boolean functions for the *next state* signals $\mathbf{s}'_{\text{visible}}$ and $\mathbf{s}'_{\text{hidden}}$, where “in X” resembles a helper signal to indicate that the FSM is currently in state X:

$$\begin{aligned}
\mathbf{s}'_{\text{visible},1} &= \text{in A or in C or in D} \\
\mathbf{s}'_{\text{visible},0} &= \text{in A or in B} \\
\mathbf{s}'_{\text{hidden},1} &= \qquad \qquad \text{in C or in D} \\
\mathbf{s}'_{\text{hidden},0} &= \text{in A or in B}
\end{aligned}$$

Comparing both signals, the hidden functionality is computed if a term from the Boolean functions of the visible functionality is excluded from the computation. This is a direct result of the selected state encodings. Therefore, we can directly implement the hidden FSM by replacing the 3-input OR-gate from the equation for $\mathbf{s}'_{\text{visible},1}$ with a camouflaged gate, where the term “in A” is connected to a dummy input.

Since the dummy inputs cannot be identified as such by a reverse engineer, he will eventually recover the state transition function $\mathbf{s}'_{\text{visible}}$, although $\mathbf{s}'_{\text{hidden}}$ is computed physically. Again, note that we were able to design both, the visible and the hidden FSM. With Doppelganger a designer is in full control of what a reverse engineer will erroneously recover.

3.2 The Doppelganger Algorithm

In the following we explain the Doppelganger algorithm in detail. To improve comprehensibility, Figure 2 shows a high-level breakdown of the algorithm’s steps applied to the example from the previous section. Our explanations follow the figure step-by-step. A more formal pseudocode description of Doppelganger can be found in Appendix A.

Inputs: The main inputs for Doppelganger are a description of the visible functionality and the hidden functionality. In detail, these functionalities are given via a set \mathcal{S} of all symbols that shall be encoded and their respective conditions that may differ for both functionalities. The set of symbols contains, e.g., FSM states or bus node names, as human-readable strings — the actual binary encodings are later chosen by Doppelganger. The conditions, \mathcal{V} for the visible and \mathcal{H} for the hidden functionality, describe under which signal combinations a specific symbol shall be output, e.g., ‘*when signal_x is 1 and current symbol is symbol A, then output symbol B*’. By providing different conditions or different output symbols in the two descriptions, the designer controls the visible and the hidden functionality. In addition, predefined encodings can also be given to Doppelganger. Figure 2a shows the precise inputs for the example from Section 3.1.

Given the inputs, Doppelganger executes two steps, namely (1) choosing suitable encodings and (2) generating the logic that enables both functionalities.

Step 1) Selecting Encodings: First, suitable encodings have to be selected for all symbols. Doppelganger assigns a standard binary encoding, i.e., for $|\mathcal{S}| = n$ symbols, all encodings have $k = \lceil \log_2(n) \rceil$ bit. If the conditions of a symbol are different from the visible functionality to the hidden functionality, Doppelganger selects encodings by following a specific set of rules. Suppose that visible symbol x has to become hidden symbol y . Encodings have to be chosen such that the encoding $E(x)$ is an *overlap* of $E(y)$ (cf. Definition 1).

Definition 1 (Overlap). Bitstring $a = (a_{l-1}, \dots, a_0)$ is an overlap of bitstring $b = (b_{l-1}, \dots, b_0)$ iff $a \neq b$ and $a_i \geq b_i \forall i$.

If the above condition is fulfilled, $E(x)$ has more 1-bits than $E(y)$ and $E(x)$ has a 1-bit wherever $E(y)$ has a 1-bit. Therefore, feeding the terms that will compute the additional 1-bits of $E(x)$ into dummy signals of camouflaged gates will result in computation of the encoding of $E(y)$.

We present two algorithms for assigning encodings, (1) a greedy algorithm and (2) an exhaustive algorithm. Encodings for symbols that are overlapped or that overlap another symbol have to be chosen with care, hence we call them *critical symbols* in the following. Both algorithms assign encodings only to said critical symbols, the encodings of the remaining states do not matter for obfuscation. Hence, for all remaining symbols, the remaining available encodings are sorted in ascending order and the first available one is assigned to each. Regardless of the algorithm, the order in which encodings have to be assigned to critical symbols is determined first: if the encoding of symbol x has to overlap the encoding of symbol y , then y has to be processed before x .

Figure 2b visualizes the encoding selection process for the example from Section 3.1 using the greedy algorithm.

Greedy Encoding Assignment: The greedy algorithm assigns a suitable encoding to each critical symbol x via the following scheme: it forms a candidate set C of encodings from all unassigned encodings that are suitable for x . This is done by testing the overlap rule for all unassigned encodings with respect to the already assigned encodings of all symbols that are overlapped by x . The algorithm then simply assigns the first encoding in C to symbol x . A formal description of the algorithm can be found in Algorithm 1.

Since the greedy algorithm simply traverses all symbols once and assigns an encoding, its complexity is $\mathcal{O}(c)$ for $c \leq n$ critical symbols. However, since it always picks the first

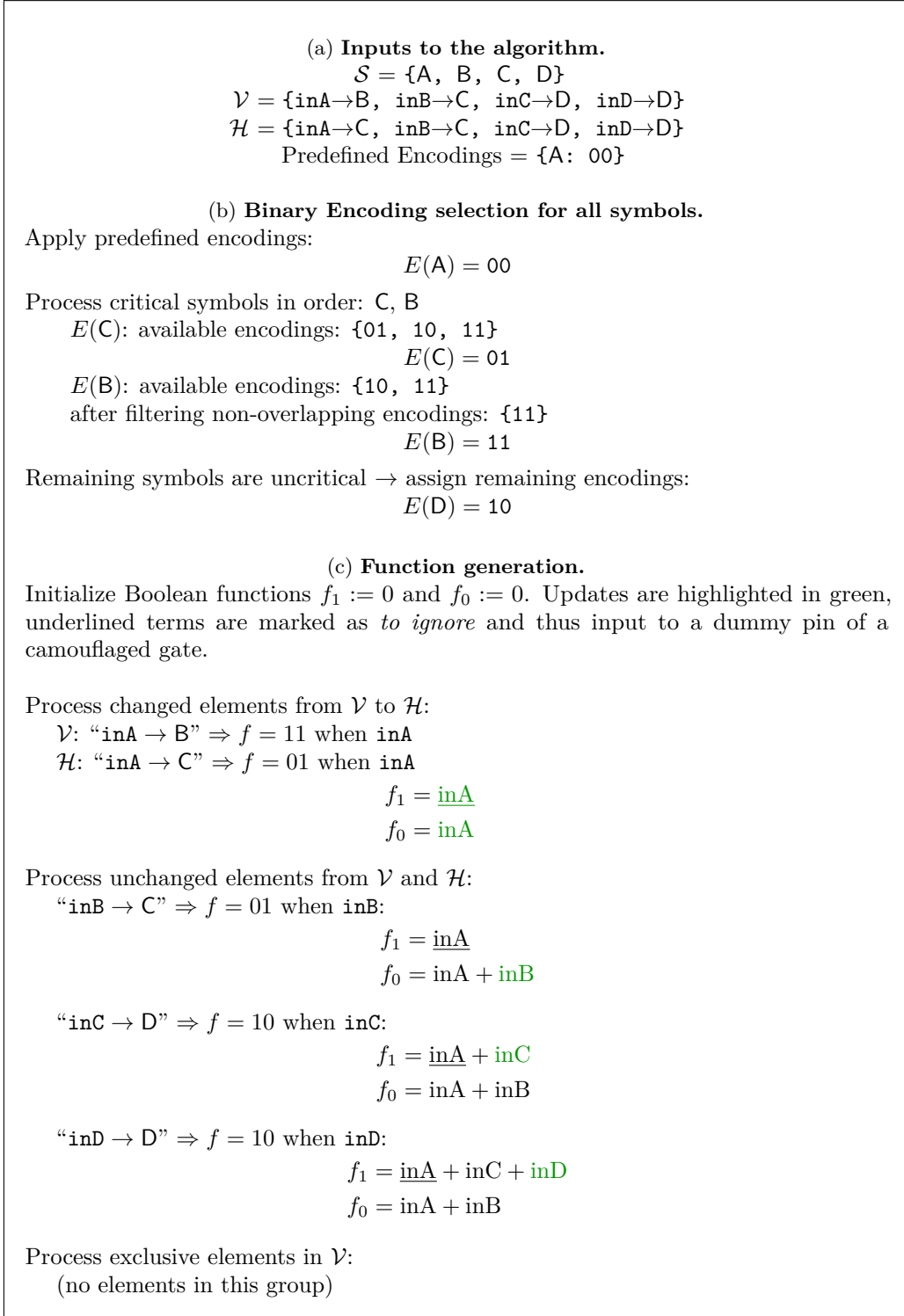


Figure 2: Rundown of the Doppelganger algorithm applied to the example from Section 3.1

suitable encoding, it can run out of candidates if the overlap structure is more complex. In that case, the exhaustive algorithm can take over, which we describe in the following.

Exhaustive Encoding Assignment: The exhaustive algorithm can be compared to a depth-first-search across all possible encoding assignments and is best described recursively. The algorithm starts with the first critical symbol and the predefined encoding assignments, if available. Now, in each recursion, the algorithm iterates over all unassigned encodings. Each encoding is checked whether it is suitable for the current symbol given the overlap rule. If the encoding is suitable, it is temporarily assigned to the current symbol and the algorithm recursively starts with the next symbol. If at any time no suitable encodings are available anymore, the current path is aborted. However, if an execution path ends by assigning a suitable encoding to each critical symbol, the algorithm terminates and returns the current encoding assignments. Instead of terminating, the algorithm can also continue and thus find all possible encoding assignments for the given inputs. A formal description of the algorithm can be found in [Algorithm 2](#).

For $c \leq n$ critical symbols, the exhaustive algorithm has a worst-case complexity of $\mathcal{O}(2^{k^c})$, which can be approximated as $\mathcal{O}(c^c)$. This can lead to long run times depending on the number of critical symbols. However, in practice, the algorithm’s runtime is a minor issue: the whole computation is only performed once in the design process. The algorithm itself can be easily parallelized in its search paths and terminates as soon as a suitable encoding for each symbol is found.

In our implementation, the greedy algorithm is used as the initial algorithm and the exhaustive algorithm only takes over if the greedy version fails. Note that if a suitable encoding assignment exists for all critical symbols, the exhaustive algorithm will definitely find it.

Step 2) Generating Obfuscated Logic: After encodings have been assigned, the obfuscated logic is generated in form of a Boolean function f_i for each output bit of the encoding. All output functions are initialized to $f_i = 0$. Doppelganger then analyzes the elements in \mathcal{H} and \mathcal{V} and assigns them to one of three groups: (1) those which are only present in \mathcal{V} , (2) those which were changed from \mathcal{H} to \mathcal{V} , and (3) those which are contained in both \mathcal{H} and \mathcal{V} . It then iterates these groups and generates the output functions bit-by-bit in DNF format, based on the encoding of the respective output symbol in the visible functionality. Depending on the assigned group, specific DNF-terms are marked as *to ignore* if they appear only in visible functionality. By connecting all signals that hold the terms marked as *to ignore* to dummy inputs of camouflaged gates, the hidden functionality is actually computed because of the overlapping encodings.

[Figure 2c](#) shows the final function generation step-by-step for the example from [Section 3.1](#). A formal description of the algorithm can be found in [Algorithm 3](#).

4 Case Study Overview

In this section, we give an overview on the cryptographic coprocessor that is used as the baseline for our two case studies in [Section 5](#) and [Section 6](#).

4.1 Motivation

The chosen design resembles a cryptographic coprocessor for symmetric ciphers as used in many applications with security features. The user is able to select from a variety of cipher primitives and modes of operation to leverage hardware-accelerated cryptography.

Our design employs a standard architecture: modularity is achieved via a centralized bus that interconnects the available functional modules and register files. Both, cipher cores and the additional functionalities of modes of operation, can thus be implemented as isolated modules, connected to the bus, and then used interchangeably. While the architecture

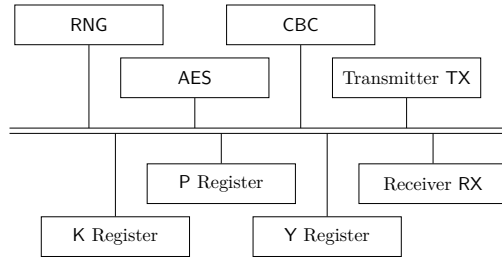


Figure 3: Functional modules of the CBC-encryption core, connected via a central data bus

allows to add an arbitrary number of cipher primitives and modes, we implemented support for one cipher (AES) and one mode of operation (CBC) for the sake of simplicity. In CBC mode, the previous output (or the IV in case of the first block) is xored to the plaintext and the result is encrypted.

4.2 Design

All functional modules and bus connections of our implementation are shown in [Figure 3](#). It consists of I/O modules RX and TX, a random number generator RNG, an AES encryption module AES, a CBC module CBC, and three registers: K for a key, P for a plaintext, and Y for the output. The modules are orchestrated via a central FSM, which also operates a Bus Controller that in turn controls the data flow. We selected a bus width of $w = 32$ bit. For simplicity, the bus connections are implemented via tri-state logic: the current master can arbitrarily write to the bus lines and all other nodes keep their bus connections on high-impedance to allow for signal transmissions in arbitrary directions. Note, that the mode-of-operation functionality, in case of CBC a simple xor, is intentionally computed outside of the cipher module. This way, additional modes are easily added, while being decoupled from the underlying cipher primitive. For the same reason, there is a dedicated register for the plaintext P, which could be merged with the output register Y if only CBC-mode had to be supported.

We synthesized the design using the Nangate 45nm standard cell library and the Synopsys design suite. Here, encodings of the components we will obfuscate, e.g., the central FSM and the bus controller addresses, were simply assigned in ascending order. The entire synthesized coprocessor has an area of 16,681 GE, comprised of 10,165 combinational cells and 1048 sequential cells. The main FSM has an area of 62 GE, comprised of 44 combinational cells and 4 sequential cells.

4.3 Functionality

In our case studies, Doppelganger will be applied to the central FSM and the bus controller. [Figure 4](#) shows the combined functionality of both components. Note that, for the sake of readability, we omitted arrows indicating a waiting state, i.e., we only included transitions to other states. Each state circle contains its functionality as well as the performed bus transmissions (if available).

The overall functionality of the design is as follows: After initialization, the core awaits a control byte indicating either a new key or a new plaintext for encryption. If a new key should be set, the core stores the new key, generates a fresh Initialization Vector (IV), stores it in the Y-register, and transmits it to the user. In case a new plaintext is received, it is stored in the P-register. Then, it is sent to the CBC-module together with the previous ciphertext from the Y-register, which is the IV in the first encryption, and stores the result back into P. This result is then sent as the plaintext to the encryption module, followed by

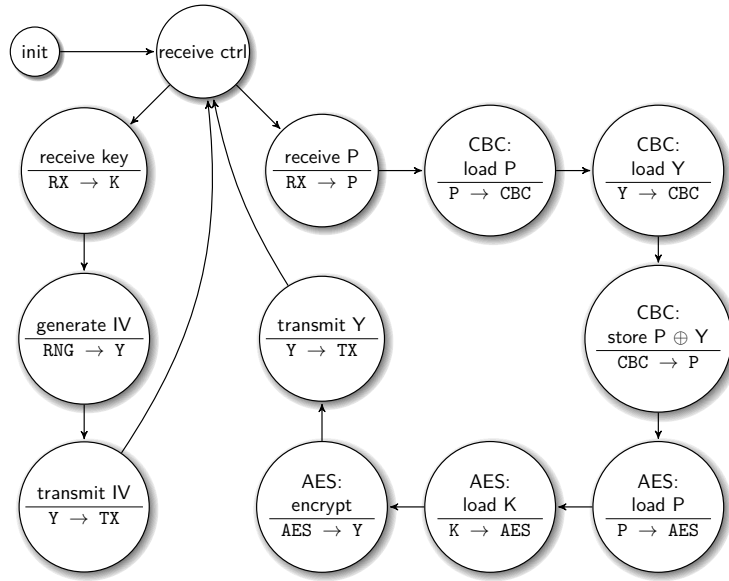


Figure 4: Functionality of the design. For readability, only transitions to other states are shown.

the key. When encryption is done, the ciphertext is stored in Y and then transmitted to the user.

Thus, this design implements a cryptographic coprocessor that offers benign AES-CBC functionality. There are no unusual design decisions and input/output behavior is as expected given the specifications. Note that a fully-fledged design would likely feature decryption functionality, additional modes of operation, and other symmetric ciphers, all of which further increase complexity and thus the required effort for reverse engineering.

5 Case Study I – Defensive Obfuscation

In our first case study, we assess the impact of applying Doppelganger as a protective measure. We show that a reverse engineer can be tricked into recovering a different but *plausible* functionality. Furthermore, we demonstrate that partially random obfuscation can massively hinder the analyst if plausibility cannot be easily verified in an application.

5.1 Plausible Obfuscation

In the case of a cryptographic coprocessor, control logic can quite easily be checked for plausibility when the functional modules are understood. In the first part of this case study, we therefore apply Doppelganger such that the visible functionality is plausible and presents the reverse engineer with a different mode of operation, namely CFB instead of CBC. In this mode, the previous output (or the IV in the first block) is encrypted and the plaintext is then xored.

To lead a reverse engineer to recover this (incorrect) mode, we applied Doppelganger to the central FSM and the bus controller of our coprocessor. The hidden functionality is simply the original functionality of the design. For the visible functionality we changed three state transitions and a bus address decoding to suggest the CFB mode, as shown in Table 2. With these changes, our design indeed first encrypts the previous output, stored in the Y -register and combines the output with the plaintext via xor by reusing the CBC functionality *afterwards*. Note that the final transition was changed to transmit IV

Table 2: Introduced changes in the visible functionality to the FSM state transitions (top) and bus address decoding (bottom)

Select Signals	Output in \mathcal{H}	Output in \mathcal{V}
state = receive P	CBC: load P	AES: load P
state = AES: encrypt	transmit Y	CBC: load P
state = CBC: store $P \oplus Y$	AES: load P	transmit IV
state = AES: load P	$P \rightarrow \text{AES}$	$Y \rightarrow \text{AES}$

instead of **transmit Y** to avoid a circular dependency between overlapping encodings. Since effectively in both states the Y-register is transmitted and the next state is **receive ctrl** for both, this results in a functional design.

We synthesized the obfuscated design using the same options as for the unobfuscated design. The synthesized obfuscated coprocessor has a total area of 16,770 GE (+0.53%), includes 10,201 combinational cells (+0.35%), and four signals have to be connected to dummy inputs of camouflaged gates. The number of sequential cells is unchanged as expected, since Doppelganger only affects combinational logic. Notably, the FSM itself only grew by 3 combinational cells in comparison to the unobfuscated design.

To assess the output of Doppelganger, we then used the netlist analysis framework HAL [FWS⁺18] to analyze the resulting FSM implementation from the point-of-view of a reverse engineer. There, we applied the recent FSM detection and state transition recovery algorithm from Fyrbiak et al. [FWD⁺18]. The recovered state transition graph contains all transitions that are possible with the given combinational logic. Note that this can include state transitions that never occur in practice, but that are generally possible when just looking at the combinational logic. The resulting state transition graph is shown in Figure 5a. For readability, we show the corresponding state names grayed-out next to the binary encodings. Note that a real-world analyst does not have these semantic names and has to recover the purpose of each state with further analyses. The algorithm from [FWD⁺18] correctly recovered the obfuscated FSM, since the transitions match our specification of the original FSM (cf. Figure 4). Furthermore it uncovered the presence of the reset signal, since every state can transition back to the initial state.

Discussion: When analyzing the visible functionality, the reverse engineer will recover a valid AES-CFB core instead of an AES-CBC core. Note that the obfuscation was introduced at a mere overhead of 36 additional combinational cells in total, only 3 of them in the FSM. If the analyst is constrained to static analysis, the plausibility of the visible functionality therefore results in an incorrect understanding of the design. However, with dynamic analysis capabilities, the analyst is able to detect a mismatch between expected and obtained results. Note that he is still not able to immediately tell that parts of the design are obfuscated, nor can he immediately locate the obfuscation that way. Only through further analysis, he may be able to identify the source of his mismatched results (cf. Section 2.2). In the end, this notably increases the effort and cost of the reversing process, rendering the obfuscation successful in any case.

5.2 Randomized Obfuscation

Arguably, the plausibility of the visible functionality of the coprocessor can be verified once the crucial functional modules are understood, i.e., after the AES core and the registers have been identified. However, this is not the case if there is no or little a-priori knowledge about the target netlist, which is often the case during reverse engineering. If plausibility cannot be easily verified, a (partially) randomized visible functionality can thus heavily impede progress of the reverse engineer. As an example, we apply such

Table 3: Randomized changes to the state transitions of the central FSM

Select Signals	Output in \mathcal{H}	Output in \mathcal{V}
state = receive ctrl and ctrl byte = new key	receive key	AES: load P
state = generate IV	transmit IV	AES: load K
state = receive P	CBC: load P	CBC: load Y
state = AES: load P	AES: load K	receive key
state = AES: load K	AES: encrypt	transmit Y
state = transmit Y	receive ctrl	CBC: store P \oplus Y

Table 4: Area comparison of the defensive applications of Doppelganger. #CC denotes the number of combinational cells.

Name	Entire Design				FSM only	
	Area		#CC		Area	#CC
Unobfuscated	16,681 GE	–	10,165	–	62 GE	44
Plausible Obf.	16,770 GE	+0.53%	10,201	+0.35%	63 GE	47
Randomized Obf.	16,774 GE	+0.56%	10,203	+0.37%	60 GE	40

randomized obfuscation to the cryptographic coprocessor. In the visible functionality, we replace six output symbols, i.e., six “next states”, with random other states. The symbolic modifications are shown in Table 3.

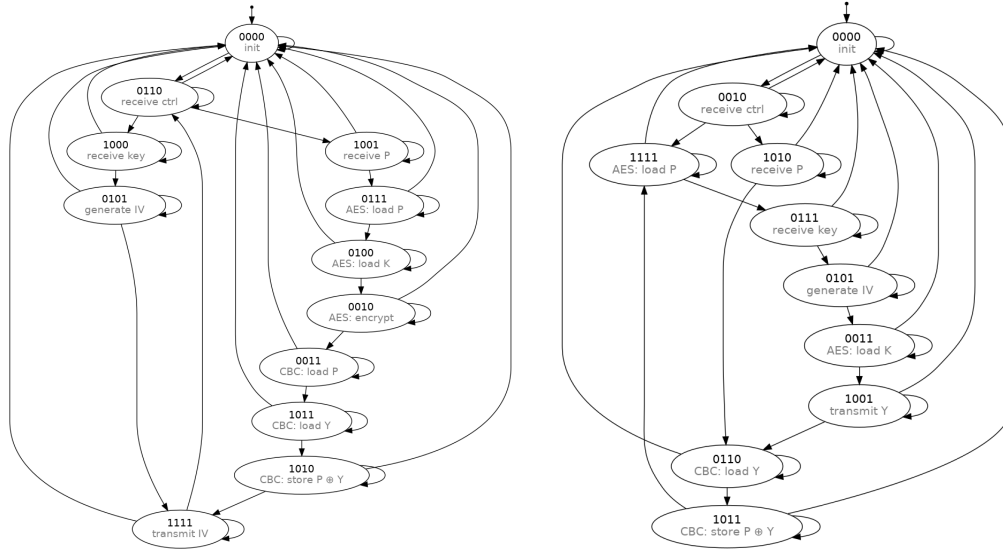
Again, we synthesized the obfuscated design using the academic Nangate 45nm standard cell library. The synthesized randomly-obfuscated coprocessor has a total area of 16,774 GE (+0.56%), includes 10,203 combinational cells (+0.37%), and 6 signals have to be connected to dummy inputs of camouflaged gates. As in the previous example, the number of sequential cells is unchanged. Table 4 shows area comparisons of the two obfuscated designs and the original synthesized design. Interestingly the FSM itself even decreased in size, however the resulting control circuitry that operates on the FSM states grew because of more complex control signal deduction.

Applying the state transition recovery algorithm to the obfuscated design results in the graph shown in Figure 5b. Comparing the recovered state transition graph with the transitions of the true functionality shows the major additional hurdles randomized obfuscation introduces for a reverse engineer who cannot check for plausibility:

First, three functional states of the hidden FSM, i.e., transmit IV, CBC: load P, and AES: encrypt, are missing in the graph, which are thus not even considered by the analyst. Note that it is common that not all possible values of the FSM state register encode valid states. The remaining 12 states still require four bits to be encoded, hence not raising suspicion.

Second, the overall structure of the recovered state transition graph does not match the original structure anymore: instead of two isolated state sequences (reacting to a new key and encrypting a plaintext) which return to a common state, the recovered state transition graph of the obfuscated FSM has two converging sequences that eventually lead to a loop.

Third, the recovered state sequences are completely different from the actually traversed sequences. Hence, even if the analyst managed to reverse engineer the isolated functionalities of the controlled modules, the recovered interworkings are *incorrect*. Assuming that plausibility of the recovered FSM cannot be easily verified, it is virtually impossible for an analyst to detect the obfuscation via static analysis. Note that even if the analyst



(a) Recovered state transition graph of the plausibly-obfuscated FSM (b) Recovered state transition graph of the randomly-obfuscated FSM

Figure 5: Recovered state transition graphs using the algorithm from [FWD⁺18]. Note that the analyst does not know the semantic meanings marked in gray.

concluded that the FSM must be obfuscated on gate level, and deduced what kind of obfuscation was in place, he would have to repeat the entire physical reversing process to eventually analyze the FSM.

Discussion: At an overhead of just 38 combinational cells, Doppelganger heavily obfuscates the central control logic. Analysis of the resulting circuitry shows that the difficulty of understanding the design is notably increased, assuming that the plausibility of the FSM cannot be verified. If the output of such a design is also probabilistic or cannot be reasonably cross-checked with simulation results, we argue that the obfuscation constitutes a major hurdle for the reverse engineer.

5.3 Evaluation of the Defensive Strength

As for the majority of obfuscation methods, it is difficult to provide quantitative measures for stealth. In the following, we provide a qualitative discussion of the stealthiness of Doppelganger.

Naturally, the strength of the obfuscation depends on the amount of information and techniques available to an analyst. In the strongest case, where an analyst cannot perform plausibility checks and cannot meaningfully check internal states of the design, e.g., because dynamic analysis is not possible or because outputs are probabilistic, even randomized application of Doppelganger is virtually impossible to detect. If the analyst has said capabilities, a carefully crafted plausible functionality of Doppelganger still results in high stealthiness. In that case, detectability solely depends on the application, as we will also demonstrate in our case study of malicious obfuscation (cf. Section 6).

Note that our obfuscation inferred almost negligible area overhead for the entire design. In the randomized example, it even led to reduction of the area of the obfuscated module.

Regarding overhead for the analyst, our case study shows that, regardless of the scenario, he has to overcome major additional obstacles. First, he has to make sure that a potential mismatch was not caused by an error during netlist recovery, which is

a very common problem [BWL⁺20]. Now, even if the analyst was sure that there was no error, he still has to narrow the source of the mismatch down to a few specific gates, which is a challenging task, especially in large IC with billions of transistors. Assuming that the analyst successfully located the obfuscation, he then has to uncover the hidden functionality, which in turn requires him to understand how the camouflaged gates are instantiated.

6 Case Study II – Stealthy Hardware Trojans

While obfuscation schemes have thus far primarily been presented as defensive techniques, they can also be employed by malicious actors. In this section, we demonstrate the dangerous implications of **malicious obfuscation**, by the example of a dangerous cryptographic backdoor. This very topic underlies the current discussion about trust in foreign-build computer and communication devices [KS]. We demonstrate that this is a viable threat by presenting an invisible, remotely exploitable, key-leakage Trojan, which is facilitated by Doppelganger. Since a reverse engineer will find a benign design and only the hidden functionality contains the malicious functionality, we call the Trojan **Evil Twin**.

6.1 Motivation

Hardware as the root of trust is often implicitly assumed to be free of manipulation. However, Trojans in hardware are a dangerous threat for high-value targets, e.g., critical infrastructure components such as network routers, the smart grid, government communication, or military systems. An example for the latter is the alleged backdoor in Syrian air defense systems, which was exploited in 2007 [Pag, Ley]. Notably, the US Department of Defense even advised against importing hardware from foreign countries because of the associated risks in its 2015-report [oD]. The 2018 Bloomberg allegations on a supposed hardware backdoor in Supermicro server hardware [RR18] and the recent ban by the US government on using telecommunication equipment from China due to security concerns [KS] show the ongoing relevance of the issue. Trojans that are near-impossible to detect are particularly attractive for nation-state actors. While common scenarios describe Trojans that are inserted by a third party, they can also be directly implanted by the original designer, e.g., when pressured by a government, as described in the Snowden documents, cf. [SFKR15].

Several sophisticated hardware Trojans have been proposed in the academic literature, ranging from side-channel Trojans [L⁺09, EGMP17], over dopant-based Trojans [B⁺13], Trojans which introduce exploitable bugs [BCS08, G⁺16] or exploit parasitic effects [KAFP19], to analog Trojan triggers in digital designs [Y⁺16]. The different classes of hardware Trojans as well as countermeasures have been surveyed in [T⁺10, CNB09].

Scenario: In this case study, we take the point-of-view of a **malicious designer**, who wants to implement a hidden backdoor in the cryptographic coprocessor from Section 4. The backdoor has to stay hidden even if the IC is reverse engineered, hence we employ Doppelganger to stealthily subvert the given design. Recall that the adversary, i.e., an analyst that reverse engineers the trojanized design, is outside of the fabrication chain and external to the design house, e.g., an inspector of foreign infrastructure technology before wide-spread deployment or an analyst during certification assessment. Given the application, the analyst is quite unrestricted in his approaches: plausibility checks and comparisons to functional ICs are possible. Therefore, the visible functionality has to be plausible with respect to the design specifications **and** the real output of the hidden functionality has to be indistinguishable from expected output under reasonable inspection. However, note that the analyst does not have access to a golden model or semantic design internals, as he is not part of the design house.

6.2 Trojanization Strategy

Following the Trojan taxonomy of [T⁺10, CNB09], Evil Twin is an *always-on* Trojan. The payload, the user-supplied symmetric key, shall be leaked via the encryption output. While symmetric encryption in isolation does not leave room for key leakage, the mode of operation can be used to exfiltrate data: most common modes, e.g., CBC, CFB, OFB, and partially CTR or GCM, use random IVs to provide their security guarantees. The IV is transmitted in plain together with the subsequently encrypted blocks. Hence, our Trojan exploits the (seemingly) random IV, for leaking the secret symmetric key. Note that the premise of such an *IV-Trojan* was sketched in [BPR14], however we craft a notably more stealthy variant that does not have to insert an adversary-chosen key into the design.

The central bus architecture, while being a common design element, is crucial to facilitate the Trojan. In order to leak the key through the IV, there has to exist a data path between the IV register and the key register. A bus structure creates a physical data path between all connected components, even though not all pairs of connected components exchange data during normal operation.

Related Work: Young and Yung introduced kleptography [YY96, YY97], inspired by covert channels [Sim84], as the study of “stealing information securely and subliminally”. The premise of a trojanized implementation in general that presents genuine input/output behavior but includes exploitable mechanisms was formalized as Algorithm Substitution Attacks (ASAs) by Bellare et al. [BPR14, B⁺15]. Note that none of these theoretical concepts have been transferred or applied to hardware designs so far. To the best of our knowledge, Evil Twin is the first time a hardware design with a kleptographic ASA Trojan has been demonstrated.

6.3 Applying Doppelganger

It is important to note that we do not make any changes to the design’s architecture to implement the Trojan. Rather, we apply Doppelganger to the central FSM and the bus controller. The visible functionality is simply the benign functionality of the coprocessor. However, the hidden functionality turns said benign design into a trojanized design.

The most naive approach to facilitate a key-leakage Trojan would be to set the IV as a plain copy of the key. However, this can be trivially detected from I/O testing, even by an unsuspecting user. Generally speaking, as long as the IV is the result of a deterministic function of just the key, the IV will be **static**. Setting the same key multiple times will always result in an identical IV which is easily noticed.

In order to overcome that limitation, the IV has to be the result of a *probabilistic* function. This leads to a delicate trade-off: The malicious designer has to be able to recover the key from a randomized IV in order to exploit the Trojan. At the same time, the entropy in the IV has to be high enough to thwart detection during testing. Therefore, we implement Evil Twin via the following strategy: Evil Twin uses the existing AES module to encrypt the user-supplied key k_{user} with an ephemeral key k_{rnd} and outputs $IV = AES_{k_{\text{rnd}}}(k_{\text{user}})$. Since an entirely random k_{rnd} would require breaking the underlying cipher to recover k_{user} , the strategy is to generate a partially random key. In our design, randomness is obtained by filling the key register with output of the RNG module. We generate a key with reduced entropy by starting encryption early, i.e., before all bus transmissions from the RNG module are completed. This results the previously reset encryption core to use a key of the binary form $k_{\text{rnd}} = 00 \dots 00|R$, with R being a random bit string of length $t \cdot w$, where w is the bus width and t the number of transmitted bus words before encryption is started. The Trojan is therefore parameterizable during design time via the security parameter $\lambda = t \cdot w$. For example, starting encryption after two transmitted data words ($t = 2$) with a bus width of $w = 16$ would result in k_{rnd} consisting of 96 zero bits followed by 32 random bits and thus allows for 2^{32} unique IVs for one static user-supplied key.

Table 5: Introduced changes in the visible functionality to the FSM state transitions (top) and bus address decoding (bottom) to enable Evil Twin. Note that this time \mathcal{V} resembles the original functionality.

Select Signals	Output in \mathcal{H}	Output in \mathcal{V}
state = receive key	transmit IV	generate IV
state = generate IV	AES: encrypt	transmit IV
state = transmit IV	generate IV	receive ctrl
state = transmit IV	K \rightarrow AES	Y \rightarrow TX
state = generate IV	RNG \rightarrow AES	RNG \rightarrow Y

Evil Twin Details: We implemented our Evil Twin Trojan by applying Doppelganger to both, the state transition logic of the central FSM and the bus addressing logic. The changes are shown in Table 5 and the resulting hidden functionality is depicted in Figure 6.

The hidden functionality operates as follows: Once a key is received in state `receive key` the FSM transitions to `transmit IV` instead of `generate IV`. This is done since we need the RNG module later, which is enabled during `generate IV`. Now in former state `transmit IV`, the bus source address and bus destination address are modified. Instead of transmitting the content of the Y register to the TX module, the content of the K register is sent to the encryption module AES as plaintext. After that, the hidden FSM transitions to `generate IV`. Here, instead of transmitting the output of the RNG module to the Y register, it is transmitted directly to the encryption core as the key. The final modified state transition starts the encryption process early, by transitioning to `AES: encrypt` directly after the first bus transmission ($t = 1$). While the remaining data of the RNG module is still transmitted to the bus, all bus read buffers are in high-impedance state, i.e., this does not impact the AES module, which now holds the value $k_{\text{rnd}} = 00 \dots 00 | R$ in its internal key register. R is the first 32-bit data word that was transmitted by the RNG module. Note that the entire right side of the functionality in Figure 6 remains unchanged in order to keep plaintext encryptions unmodified. In the following cycles, AES finishes, outputting $AES_{k_{\text{rnd}}}(k_{\text{user}})$ into the Y register, which is subsequently transmitted to the user in `transmit Y`. Therefore, the user receives a seemingly random value as the IV, which will be used for encryption of the next transmitted plaintext.

Exploiting the Trojan: During operation, the user key can be recovered from just a single wiretapped transmission that includes the IV and the encryption of a single block of (partially) known plaintext. In practice, parts of a plaintext are often known, e.g., formatting information or protocol header data. In order to recover k_{user} , the IV is decrypted while iterating through all possible k_{rnd} , yielding candidate keys. If decrypting the ciphertext with a candidate key matches the known plaintext, k_{user} was successfully exfiltrated. This key search requires on average $2^{\lambda-1}$ operations, consisting of decryption of the IV and decryption of a ciphertext with the resulting key candidate. Hence the average cost of recovering the user key is 2^{λ} decryptions. The entropy of k_{rnd} can easily be adapted by changing either w or t . However, this also scales the exploitation complexity accordingly. In our implementation with $w = 32$ and $t = 1$, the Trojan is exploitable via 2^{32} AES decryptions on average.

6.4 Evaluation of Evil Twin

Overhead: We synthesized the trojanized design with the same options as in our first case study Section 5. The synthesized trojanized coprocessor has a total area of 16,683 GE (+0.01%), includes 10,165 combinational cells ($\pm 0\%$), and six signals have to be connected to dummy inputs of camouflaged gates. Note that the number of combinational cells in

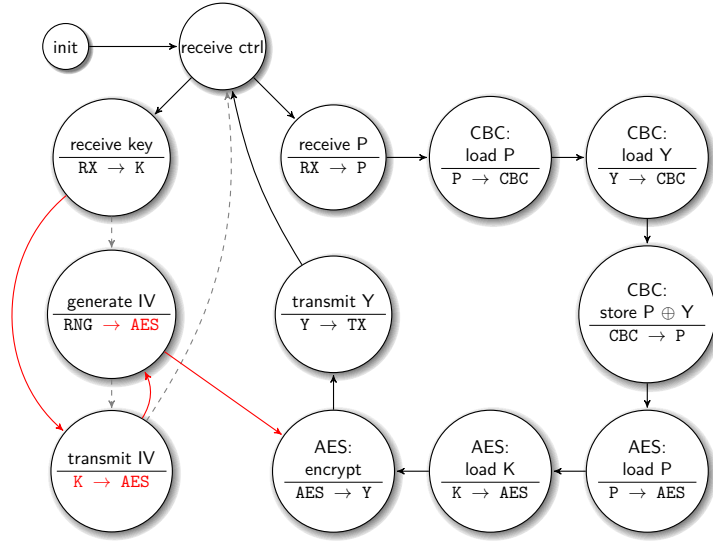


Figure 6: Overview on the realization of Evil Twin via Doppelganger. The gray transitions of the visible functionality are changed to the red ones in the hidden functionality. Likewise, bus address modifications are shown in red.

the entire design but also in the FSM specifically did not change in comparison to the original design. The FSM in isolation was even reduced in size to 60 GE (from 62 GE in the original design) due to the usage of smaller combinational cells enabled by the different encodings. Again, the number of sequential cells is unchanged as expected. The FSM analysis algorithm from [FWD⁺18] correctly recovered the visible functionality.

Trojan Strength: Intuitively, a hardware Trojan requires additions to the benign design, e.g., new functional modules, additional gates, or even analog components. Such additional components are visible in the netlist and make for suitable anchor points during reverse engineering-based inspection [BFS15]. Crucially, all existing static analysis methods for Trojan detection, e.g., [WSS13, OSYT15, HYT17, FWS⁺18], fail when faced with our Trojan, since the analyzed netlist perfectly describes the visible, benign functionality and malicious functionality is achieved by *ignoring* circuitry, not by adding new logic. Given that the analyst can employ dynamic analysis, it is possible, however highly unlikely, that the presence of our Trojan can be detected. While we cannot assess all available dynamic analysis approaches, in the following we argue that the Trojan stays undetected for well known techniques.

Since the IV is (partially) random, netlist-based test pattern generation, e.g., via Automated Test Pattern Generation (ATPG), is not applicable. Regarding latency, the trojanized design takes longer to output an IV since a full encryption has to be performed. Hence, comparing the expected duration of IV generation with the duration on a functional IC might reveal a mismatch. However, this is only the case if there are no unpredictable delays, e.g., when querying a TRNG. If the obfuscation is paired with common side-channel countermeasures such as random delays, this analysis does not reveal the Trojan. Likewise, increased power consumption during IV generation is only visible when comparing with a golden model, which the analyst does not have.

Looking at the Trojan design, the most promising approach for detection lies in finding IV collisions: if for a fixed key the same IV is observed multiple times in short periods, suspicion arises. Based on the birthday paradox such a collision is expected after approximately $2^{\lambda/2}$ attempts for a fixed user key. Instantiating the design with $\lambda = 32$, the first collision is hence expected after roughly 2^{16} IVs. However, since there is no indication

of a Trojan at any point of the reversing process, it is not reasonable to observe that many IVs for a single static key out of curiosity. Recall that in order to exploit the Trojan, 2^{32} decryptions are required on average. Such a computation can be done in a matter of minutes on a modern desktop PC purely in software and even faster using the specialized AES-NI instructions.

In total, we expect Evil Twin to remain undetected. In a real-world implementation, with several more functionalities and potentially countermeasures against other attack vectors, the Trojan would be even more stealthy than in our experimental setting. Note that, to the unsuspecting user, the Trojan is entirely invisible since IVs appear to be random and the device is interoperable with genuine devices.

Parametrization: In order to further decrease the likelihood of detectability from testing IV collisions, the security parameter λ can be increased. However, this negatively impacts exploitability, as one has to invest exponentially more computations to recover a user key. For example, at $\lambda = 64$ the analyst has to observe an average of 2^{32} IVs for the same key before a collision is expected, while the Trojan designer would need to perform 2^{64} decryptions to recover the user key. Even though this is considerable effort, it seems highly likely that nation-state adversaries can routinely perform such computations. Hence, the trade-off between detectability and exploitability also introduces a notion of exclusivity: if the workload to exploit the Trojan is increased, only well-equipped actors can exploit the Trojan even after it is uncovered.

7 Discussion

After demonstrating Doppelganger in two case studies and discussing its strength with respect to the scenarios, we will now discuss its overall overhead and limitations.

7.1 Overhead

Typically, obfuscation schemes introduce a quantifiable overhead, e.g., by increasing the design area by a fixed percentage. As shown in our case studies, Doppelganger comes at negligible overhead in terms of area increase, even when only analyzing the obfuscated areas. In general, the overhead grows with the difference between the visible and the hidden functionality, i.e., the number of critical symbols and changed select signals. The more terms differ in the Boolean output functions, the more combinational logic is effectively ignored by camouflaged gates, hence becoming dummy logic. However, a more precise generic quantification is difficult to provide, since the output of the obfuscation, i.e., the visible functionality, is fully under the designer's control. Still, encoding logic, i.e., the target for Doppelganger, typically makes an integral but area-wise very small part of modern ICs as also shown in our case studies. Thus, the overhead for the whole design is expected to be extremely low in virtually all applications.

7.2 Limitations

Doppelganger is a valuable defensive tool for design obfuscation (cf. Section 5) and at the same time poses a considerable threat in offensive applications (cf. Section 6). It should be noted that Doppelganger can easily be combined with traditional obfuscation methods, e.g., bus scrambling. However, like all obfuscation it comes with limitations.

First, the stealthiness of Doppelganger is dependent on the strength of the underlying low-level obfuscation. Should an analyst be able to uncover the camouflaged gates, he can recover an error-free netlist of the IC's hidden functionality and can continue with regular design reverse engineering. If he is able to do so immediately during the standard netlist recovery process (cf. Section 2.1), i.e., the camouflaged gates are broken, Doppelganger is

broken as well if instantiated with said gates. However, note that this is caught in our requirements for successful application of Doppelganger that were presented in Section 3.

Another limitation lies in the fact that Doppelganger is only applied to a small part of the design. While encoding logic is typically present in the central elements that orchestrate the majority of a design, in many applications a reverse engineer can still correctly analyze other functional modules, e.g., the modules in our case study shown in Figure 3. However, in more complex designs with a multitude of shared functional components, understanding the control logic is imperative to understand the high-level functionalities of the design.

A potential way to quickly detect and locate low-level obfuscation in general is via scan chains, where the state of each flip-flop in the IC can be written and read out. Virtually all ICs are equipped with (partial) scan chains for testing, however, they are typically deactivated after fabrication, especially in security-critical products. Thus, it is imperative that re-activation of the scan chain is not feasible.

A minor drawback of Doppelganger is that it puts constraints on the synthesis process. Since specific signals have to be input to camouflaged gates, they must not be removed or merged during synthesis optimization, which can lead to locally slightly less optimized logic. However, since optimization is typically done using heuristics and highly depends on the synthesis targets, e.g., optimize for area, these minor inefficiencies are common and thus do not raise suspicion, as emphasized by the minuscule overhead in our case studies.

8 Conclusion

In this work we presented Doppelganger, a novel and generic hardware obfuscation technique for arbitrary encoding logic. Built on camouflaged gates, the designer not only has control over the functionality that is obfuscated, but also over the output a reverse engineer is facing. This is a unique feature among obfuscation schemes in general and particularly attractive when obfuscating complex designs.

We demonstrated the defensive strength of Doppelganger by the example of a cryptographic coprocessor at an area overhead of less than 0.6%. Especially when the visible functionality is carefully crafted, a strongly obfuscated design can be achieved. In our second case study, we demonstrated for the first time that obfuscation with camouflaged gates is a double-edged sword. We show how a kleptographic Algorithm Substitution Attack (ASA), initially proposed as a theoretical construct, can in fact be stealthily mounted using Doppelganger. The trojanized design — Evil Twin — is still interoperable with genuine designs and allows for remote key-leakage via a known-plaintext attack, again at negligible overhead. Evil Twin allows eavesdropping on application-level communication, even though the attack is introduced at chip-level, i.e., unaware of said subverted application.

In summary, Doppelganger offers major advantages over the state of the art in hardware obfuscation, which is almost entirely restricted to traditional high-level obfuscation methods. It is very stealthy and introduces minuscule overhead while notably impeding reverse engineering. On the other hand, we demonstrated that low-level obfuscation enables the novel threat of hidden Trojans within seemingly entirely benign functionality. Crucially, even though a large body of research exists on detection techniques for hardware Trojans in general, cf. [CNB09, T⁺10], they all fail against the manipulations enabled by Doppelganger-like low-level algorithms. It can be concluded that ensuring and verifying trust in hardware is even more complex than assumed thus far. Hence, our work also highlights the need for novel interdisciplinary approaches to efficiently detect stealthy modifications.

References

- [B⁺13] G. T. Becker et al. Stealthy Dopant-Level Hardware Trojans. In CHES, pages 197–214. Springer, 2013.
- [B⁺15] M. Bellare et al. Mass-Surveillance Without the State: Strongly Undetectable Algorithm-Substitution Attacks. In ACM CCS, pages 1431–1440, 2015.
- [BCCJ98] James P Baukus, Lap Wai Chow, and William M Clark Jr. Digital circuit with transistor geometry and channel stops providing camouflage against reverse engineering, July 21 1998. US Patent 5,783,846.
- [BCS08] Eli Biham, Yaniv Carmeli, and Adi Shamir. Bug Attacks. In Annual International Cryptology Conference, pages 221–240. Springer, 2008.
- [BFS15] Chongxi Bao, Domenic Forte, and Ankur Srivastava. On Reverse Engineering-Based Hardware Trojan Detection. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(1):49–57, 2015.
- [BPR14] Mihir Bellare, Kenneth G Paterson, and Phillip Rogaway. Security of Symmetric Encryption Against Mass Surveillance. In International Cryptology Conference, pages 1–19. Springer, 2014.
- [BWL⁺20] Ulbert J. Botero, Ronald Wilson, Hangwei Lu, Mir Tanjidur Rahman, Mukhil A. Mallaiyan, Fatemeh Ganji, Navid Asadizanjani, Mark M. Tehranipoor, Damon L. Woodard, and Domenic Forte. Hardware Trust and Assurance through Reverse Engineering: A Survey and Outlook from Image Analysis and Machine Learning Perspectives, 2020.
- [CBCJ07] Lap-Wai Chow, James P Baukus, and William M Clark Jr. Integrated circuits protected against reverse engineering and method for fabricating the same using an apparent metal contact line terminating on field oxide, November 13 2007. US Patent 7,294,935.
- [CBW⁺12] Ronald P Cocchi, James P Baukus, Bryan J Wang, Lap Wai Chow, and Paul Ouyang. Building block for a secure cmos logic cell library, February 7 2012. US Patent 8,111,089.
- [CBWC12] Lap Wai Chow, James P Baukus, Bryan J Wang, and Ronald P Cocchi. Camouflaging a standard cell based integrated circuit, April 3 2012. US Patent 8,151,235.
- [CC90] Elliot J. Chikofsky and James H Cross. Reverse Engineering and Design Recovery: A Taxonomy. IEEE software, 7(1):13–17, 1990.
- [CEMG16] Maria I Mera Collantes, Mohamed El Massad, and Siddharth Garg. Threshold-Dependent Camouflaged Cells to Secure Circuits Against Reverse Engineering Attacks. In VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on, pages 443–448. IEEE, 2016.
- [CNB09] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware Trojan: Threats and Emerging Solutions. In High Level Design Validation and Test Workshop, 2009. HLDVT 2009. IEEE International, pages 166–171. IEEE, 2009.

- [EEAM16] Burak Erbagci, Cagri Erbagci, Nail Etkin Can Akkaya, and Ken Mai. A Secure Camouflaged Threshold Voltage Defined Logic Family. In 2016 IEEE International symposium on hardware oriented security and trust (HOST), pages 229–235. IEEE, 2016.
- [EGMP17] Maik Ender, Samaneh Ghandali, Amir Moradi, and Christof Paar. The First Thorough Side-Channel Hardware Trojan. In International Conference on the Theory and Application of Cryptology and Information Security, pages 755–780. Springer, 2017.
- [FWD⁺18] Marc Fyrbiak, Sebastian Wallat, Jonathan Déchelotte, Nils Albartus, Sinan Böcker, Russell Tessier, and Christof Paar. On the Difficulty of FSM-based Hardware Obfuscation. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018(3):293–330, Aug. 2018.
- [FWS⁺18] Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Sebastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier, and Christof Paar. HAL-The Missing Piece of the Puzzle for Hardware Reverse Engineering, Trojan Detection and Insertion. IEEE Transactions on Dependable and Secure Computing, 2018.
- [G⁺16] S. Ghandali et al. A Design Methodology for Stealthy Parametric Trojans and Its Application to Bug Attacks. In CHES, pages 625–647. Springer, 2016.
- [HYT17] Kento Hasegawa, Masao Yanagisawa, and Nozomu Togawa. Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier. In 2017 IEEE International Symposium on Circuits and Systems (ISCAS), pages 1–4. IEEE, 2017.
- [KAFF19] Christian Kison, Omar Mohamed Awad, Marc Fyrbiak, and Christof Paar. Security implications of intentional capacitive crosstalk. IEEE Transactions on Information Forensics and Security, 14(12):3246–3258, 2019.
- [KS] Cecilia Kang and David E. Sanger. Trump’s Ban On Telecom Hits Huawei. <https://www.nytimes.com/2019/05/15/business/huawei-ban-trump.html>. Accessed: 2019-06-18.
- [L⁺09] L. Lin et al. Trojan Side-Channels: Lightweight Hardware Trojans Through Side-Channel Engineering. In CHES, pages 382–395. Springer, 2009.
- [L⁺13] W. Li et al. WordRev: Finding Word-Level Structures in a Sea of Bit-Level Gates. In IEEE HOST, pages 67–74, 2013.
- [Ley] John Leyden. Israel suspected of ‘hacking’ syrian air defences. https://www.theregister.co.uk/2007/10/04/radar_hack_raid/. Accessed: 2018-11-14.
- [LSM⁺17] Meng Li, Kaveh Shamsi, Travis Meade, Zheng Zhao, Bei Yu, Yier Jin, and David Z Pan. Provably secure camouflaging strategy for ic protection. IEEE transactions on computer-aided design of integrated circuits and systems, 2017.
- [M⁺16a] T. Meade et al. Gate-Level Netlist Reverse Engineering for Hardware Security: Control Logic Register Identification. In IEEE ISCAS, pages 1334–1337, 2016.
- [M⁺16b] T. Meade et al. Netlist Reverse Engineering for High-Level Functionality Reconstruction. In ASP-DAC, pages 655–660, 2016.

- [MBPB15] Shweta Malik, Georg T Becker, Christof Paar, and Wayne P Burses. Development of a Layout-Level Hardware Obfuscation Tool. In *VLSI (ISVLSI), 2015 IEEE Computer Society Annual Symposium on*, pages 204–209. IEEE, 2015.
- [oD] U.S.D. of Defense. Defense science board task force on high performance microchip supply. https://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf. Accessed: 2015, now offline.
- [OSYT15] Masaru Oya, Youhua Shi, Masao Yanagisawa, and Nozomu Togawa. A score-based classification method for identifying hardware-trojans at gate-level netlists. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 465–470. IEEE, 2015.
- [Pag] Lewis Page. Cyber strike and robot weapons: Can the uk dominate the fifth domain of war? <https://arstechnica.com/information-technology/2015/12/cyber-strike-and-robot-weapons-can-the-uk-dominate-the-fifth-domain-of-war/>. Accessed: 2018-11-14.
- [PAKS17] Satwik Patnaik, Mohammed Ashraf, Johann Knechtel, and Ozgur Sinanoglu. Obfuscating the interconnects: Low-cost and resilient full-chip layout camouflaging. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 41–48. IEEE, 2017.
- [QCF+16] Shahed E Quadir, Junlin Chen, Domenic Forte, Navid Asadizanjani, Sina Shahbazmohamadi, Lei Wang, John Chandy, and Mark Tehranipoor. A Survey on Chip to System Reverse Engineering. *JETC*, 13(1):1–34, 2016.
- [Ram] Rambus. SypherMedia Library (SML) Circuit Camouflage Technology. <https://www.rambus.com/security/cryptofirewall-cores/circuit-camouflage-technology/>. Accessed: 20.01.2020.
- [RR18] Jordan Robertson and Michael Riley. The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies. <https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies>, 2018.
- [RSSK13] Jeyavijayan Rajendran, Michael Sam, Ozgur Sinanoglu, and Ramesh Karri. Security Analysis of Integrated Circuit Camouflaging. In *ACM CCS*, pages 709–720, 2013.
- [S+12] Y. Shi et al. Extracting Functional Modules From Flattened Gate-Level Netlist. In *ISCIT*, pages 538–543, 2012.
- [Sem08] Semiconductor Equipment and Materials International (SEMI) Association. Innovation at Risk: Intellectual Property Challenges and Opportunities, June 2008. White Paper.
- [SFKR15] Bruce Schneier, Matthew Fredrikson, Tadayoshi Kohno, and Thomas Ristenpart. Surreptitiously weakening cryptographic systems. *Cryptology ePrint Archive*, Report 2015/097, 2015. <https://eprint.iacr.org/2015/097>.
- [Sim84] Gustavus J Simmons. The Prisoners’ Problem and the Subliminal Channel. In *Advances in Cryptology*, pages 51–67. Springer, 1984.

- [SSF⁺14] Takeshi Sugawara, Daisuke Suzuki, Ryoichi Fujii, Shigeaki Tawa, Ryohei Hori, Mitsuru Shiozaki, and Takeshi Fujino. Reversing Stealthy Dopant-Level Circuits. In International Workshop on Cryptographic Hardware and Embedded Systems, pages 112–126. Springer, 2014.
- [SSTF19] Bicky Shakya, Haoting Shen, Mark Tehranipoor, and Domenic Forte. Covert Gates: Protecting Integrated Circuits with Undetectable Camouflaging. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(3):86–118, May 2019.
- [T⁺10] M. Tehranipoor et al. A Survey of Hardware Trojan Taxonomy and Detection. IEEE Design & Test of Computers, 27(1):10–25, 2010.
- [TJ11] Randy Torrance and Dick James. The State-Of-The-Art in Semiconductor Reverse Engineering. In Proceedings of the 48th Design Automation Conference, pages 333–338. ACM, 2011.
- [VPH⁺17] Arunkumar Vijayakumar, Vinay C Patil, Daniel E Holcomb, Christof Paar, and Sandip Kundu. Physical Design Obfuscation of Hardware: A Comprehensive Investigation of Device and Logic-Level Techniques. IEEE Trans. Information Forensics and Security, 12(1):64–77, 2017.
- [WSS13] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. Fanci: identification of stealthy malicious logic using boolean functional analysis. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 697–708, 2013.
- [Y⁺16] K. Yang et al. A2: Analog Malicious Hardware. In IEEE Symposium on Security and Privacy, pages 18–37, 2016.
- [YY96] Adam Young and Moti Yung. The Dark Side of “Black-Box” Cryptography Or: Should We Trust Capstone? In Annual International Cryptology Conference, pages 89–103. Springer, 1996.
- [YY97] Adam Young and Moti Yung. Kleptography: Using Cryptography Against Cryptography. In Eurocrypt, volume 97, pages 62–74. Springer, 1997.

A Algorithms of Doppelganger

In this section we provide detailed algorithmic descriptions to complement our explanations in Section 3.2. The set of conditions of each element x in \mathcal{V} and \mathcal{H} is addressed by “ x .conditions” and the respective resulting output symbol is addressed by “ x .symbol”. Starting with the selection of suitable encodings, Algorithm 1 shows the greedy algorithm and Algorithm 2 shows the exhaustive variant. Generation of the output functions is shown in Algorithm 3

Algorithm 1 Greedy Encoding Generation

Input: Symbols \mathcal{S} , visible functionality \mathcal{V} , hidden functionality \mathcal{H} , predefined encodings E^*

Output: Encodings E or error symbol \perp

- 1: $k \leftarrow \lceil \log_2(|\mathcal{S}|) \rceil$
- 2: $E_{avail} \leftarrow \{0, \dots, 2^k - 1\} \setminus \{\text{encodings in } E^*\}$
- 3: $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\text{symbols in } E^*\}$
- 4: $E \leftarrow E^*$
- 5: $\mathcal{S}_{crit} \leftarrow \text{compute_critical_symbol_order}(\mathcal{S}, \mathcal{V}, \mathcal{H})$
- 6: **for all** $s \in \mathcal{S}_{crit}$ **do**
- 7: $\mathcal{S}_{ov} \leftarrow \text{get_overlapped_symbols}(s, \mathcal{V}, \mathcal{H})$ // get all symbols that s overlaps
- 8: $E_{ov} \leftarrow \{E(x) \mid \forall x \in \mathcal{S}_{ov}\}$
- 9: $C \leftarrow \text{get_suitable_encodings}(E_{avail}, E_{ov})$
- 10: **if** $C = \emptyset$ **then return** \perp // no suitable encodings left
- 11: $E(s) \leftarrow C[0]$
- 12: $E_{avail} \leftarrow E_{avail} \setminus \{E(s)\}$
- 13: **for all** $s \in \mathcal{S} \setminus \mathcal{S}_{crit}$ **do** // assign remaining encodings to uncritical symbols
- 14: $E(s) \leftarrow E_{avail}[0]$
- 15: $E_{avail} \leftarrow E_{avail} \setminus \{E(s)\}$
- 16: **return** E

Algorithm 2 Exhaustive Encoding Generation

Input: Symbols \mathcal{S} , visible functionality \mathcal{V} , hidden functionality \mathcal{H} , predefined encodings E^*

Output: Encodings E or error symbol \perp

- 1: $k \leftarrow \lceil \log_2(|\mathcal{S}|) \rceil$
- 2: $E_{avail} \leftarrow \{0, \dots, 2^k - 1\} \setminus \{\text{encodings in } E^*\}$
- 3: $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\text{symbols in } E^*\}$
- 4: $\mathcal{S}_{crit} \leftarrow \text{compute_critical_symbol_order}(\mathcal{S}, \mathcal{V}, \mathcal{H})$
- 5: $E \leftarrow \text{RECURSIVEWALK}(0, E^*)$ // start recursion at the first symbol in order with only the predefined encodings
- 6: **for all** $s \in \mathcal{S} \setminus \mathcal{S}_{crit}$ **do** // assign remaining encodings to uncritical symbols
- 7: $E(s) \leftarrow E_{avail}[0]$
- 8: $E_{avail} \leftarrow E_{avail} \setminus \{E(s)\}$
- 9: **return** E
- 10: **procedure** $\text{RECURSIVEWALK}(\text{index } i, \text{encodings } E')$
- 11: **if** $i \geq |\mathcal{S}_{crit}|$ **then return** E'
- 12: $s \leftarrow \mathcal{S}_{crit}[i]$
- 13: $\mathcal{S}_{ov} \leftarrow \text{get_overlapped_symbols}(s, \mathcal{V}, \mathcal{H})$ // get all symbols that s overlaps
- 14: $E'_{avail} \leftarrow E_{avail} \setminus \{\text{encodings in } E'\}$
- 15: $C \leftarrow \text{get_suitable_encodings}(E'_{avail}, E_{ov})$
- 16: **for all** $e \in C$ **do**
- 17: $E'(s) = e$
- 18: $R \leftarrow \text{RECURSIVEWALK}(i + 1, \text{copy } E')$
- 19: **if** $R \neq \perp$ **then return** R
- 20: **return** \perp

Algorithm 3 Output Logic Generation

Input: visible functionality \mathcal{V} , hidden functionality \mathcal{H} , encodings E
Output: Boolean functions f_i and corresponding sets of terms I_i marked as *to ignore*, or error symbol \perp

- 1: $n \leftarrow$ bitlength of encodings in E
- 2: $U \leftarrow \mathcal{V} \cap \mathcal{H}$ // unchanged entries
- 3: $A \leftarrow \mathcal{V} \setminus \mathcal{H}$ // entries added in \mathcal{V}
- 4: $\mathcal{H} \leftarrow \mathcal{H} \setminus U$
- 5: $\mathcal{V} \leftarrow \mathcal{V} \setminus (U \cup A)$
- 6: $C \leftarrow \emptyset$ // collect pairs of changed entries in C
- 7: **for all** $h \in \mathcal{H}$ **do**
- 8: **for all** $v \in \mathcal{V}$ **do**
- 9: **if** $h.\text{symbol} = v.\text{symbol}$ **and** $h.\text{conditions} \subset v.\text{conditions}$ **then** // conditions were changed
- 10: $C \leftarrow C \cup \{(h, v)\}$
- 11: $\mathcal{H} \leftarrow \mathcal{H} \setminus \{h\}$
- 12: $\mathcal{V} \leftarrow \mathcal{V} \setminus \{v\}$
- 13: **for all** $h \in \mathcal{H}$ **do**
- 14: **for all** $v \in \mathcal{V}$ **do**
- 15: **if** $h.\text{symbol} \neq v.\text{symbol}$ **and** $h.\text{conditions} \subseteq v.\text{conditions}$ **then** // at least the output symbol was changed
- 16: $C \leftarrow C \cup \{(h, v)\}$
- 17: $\mathcal{H} \leftarrow \mathcal{H} \setminus \{h\}$
- 18: $\mathcal{V} \leftarrow \mathcal{V} \setminus \{v\}$
- 19: // At this point, \mathcal{V} and \mathcal{H} are separated in three groups U , A , and C
- 20: **if** $\mathcal{H} \neq \emptyset$ **then return** \perp // Error: functionality in \mathcal{H} is not a subset of functionality in \mathcal{V}
- 21: **for all** $i \in \{0, \dots, n-1\}$ **do** // assign remaining encodings to uncritical symbols
- 22: $f_i \leftarrow 0$
- 23: $I_i \leftarrow \emptyset$
- 24: **for all** $x \in U$ **do**
- 25: **if** $E(x.\text{symbol})_i = 1$ **then** // check if i -th bit is set
- 26: $f_i \leftarrow f_i + \prod x.\text{conditions}$ // add condition term to DNF
- 27: **for all** $(h, v) \in C$ **do**
- 28: **if** $E(h.\text{symbol})_i = 1$ **or** $E(v.\text{symbol})_i = 1$ **then** // check if i -th bit is set in any of the output symbols
- 29: $t_h \leftarrow \prod h.\text{conditions}$
- 30: $t_v \leftarrow \prod (v.\text{conditions} \setminus h.\text{conditions})$ // extract conditions that are ignored in the hidden functionality
- 31: $f_i \leftarrow f_i + (t_h \cdot t_v)$ // add both condition terms to DNF
- 32: $I_i \leftarrow I_i \cup \{t_v\}$ // mark condition term t_v as *to be ignored*
- 33: **for all** $x \in A$ **do**
- 34: **if** $E(x.\text{symbol})_i = 1$ **then** // check if i -th bit is set
- 35: $t \leftarrow \prod x.\text{conditions}$
- 36: $f_i \leftarrow f_i + t$ // add condition term to DNF
- 37: $I_i \leftarrow I_i \cup \{t\}$ // mark condition term as *to be ignored*
- 38: **return** $f_i, I_i \forall i$
