Stealthy Opaque Predicates in Hardware - Obfuscating Constant Expressions at Negligible Overhead

Max Hoffmann and Christof Paar

Horst Görtz Institute for IT-Security, Ruhr-Universität Bochum, Germany,
{max.hoffmann,christof.paar}@rub.de

Abstract. Opaque predicates are a well-established fundamental building block for software obfuscation. Simplified, an opaque predicate implements an expression that provides constant Boolean output, but appears to have dynamic behavior for static analysis. Even though there has been extensive research regarding opaque predicates in software, techniques for opaque predicates in hardware are barely explored. In this work, we propose a novel technique to instantiate opaque predicates in hardware, such that they (1) are resource-efficient, and (2) are challenging to reverse engineer even with dynamic analysis capabilities. We demonstrate the applicability of opaque predicates in hardware for both, protection of intellectual property and obfuscation of cryptographic hardware Trojans. Our results show that we are able to implement stealthy opaque predicates in hardware with minimal overhead in area and no impact on latency.

Keywords: Opaque Predicates · Obfuscation · Hardware Reverse Engineering · Hardware Analysis

1 Introduction

The process of designing, implementing and testing hardware cores is both time consuming and costly. Thus, in today's hardware design process, not every piece of a product is created anew. Many products consist partially of Intellectual Property (IP)-cores, i.e. third party modules. The market for IP-cores is large with USD 2.45 billion in 2013 [Cla14] and was recently "expected to be worth USD 6.22 billion by 2023" [rm:17]. IP-cores are sold as either *soft-cores* in the form of HDL code, *firm-cores* as a netlist, or *hard-cores* as a fixed layout for a certain technology. Preferably they are sold as hard- or firm-cores with only the interfaces and functional specifications known to maintain exclusivity and hinder IP-theft. Consequently, the IP-core market suffers from copyright infringements as code can illegally be copied without relicensing. Additionally, IP-cores have become a major target for reverse engineering in order to understand technologies of competitors and to find hidden backdoors.

Protecting hardware designs has therefore moved into the focus of the design industry. To counter IP-theft, vendors include watermarks into their products. A watermark is a vendor- or customer-specific attribute, embedded into the design. It does not impact functionality but its existence can easily be verified by the vendor. Consequently, the vendor is able to analyze products on the market for license violations. However, since such a watermark is often a static part of the design (c.f. [CPG$^+$06, SZT08]) or mostly disconnected from the rest of the circuitry (c.f. [FT03]), it can be found by leveraging these properties and subsequently removed by reverse engineers. To counter such removal

attacks and harden against reverse engineering in general, designers can use obfuscation methods.

Obfuscation in general transforms a product to hide implementation details from reverse engineers while providing functional equivalence. However, obfuscation often negatively impacts performance. For example, Chakraborty et al. applied their technique HARPOON [CB09] to the ISCAS-89 benchmark suite [isc], which increased area by 12.44% and power consumption by 12.81% on average. Since these penalties influence sales, either weak methods are used or no obfuscation is employed at all, with designers arguing that the size of the design makes reversing already infeasible. Strong obfuscation techniques which can be implemented at negligible costs and with minimal effort could incentivize designers to protect their designs, effectively reducing the amount of information which is plainly available to attackers.

A basic building block in software obfuscation is called *opaque predicate*. An opaque predicate realizes a function that provides a constant Boolean output known to the designer, but which is difficult to deduce during static analysis. Opaque predicates can be used, for example, to harden control flow analysis. However, in hardware obfuscation, such a building block is still missing. To the best of our knowledge, there has been only a single prior work on dedicated hardware opaque predicates which was shown to be easily detectable and removable (c.f. Section 2.3).

Many hardware designs incorporate constant control or data signals on HDL level that are worth of protection (e.g. for nominal-actual comparison in sensors or cryptographic key material). Even though such constant signals may be partially merged into surrounding circuitry by optimization during synthesis, fragmentary information resides which can be leveraged for reverse engineering. For example, comparison circuitry that evaluates to *true/false* for only one input can be realized with fewer gates than a comparator that is *true/false* for more than one input, which allows a reverse engineer for easier detection and logical function recovery. Thus, obfuscating valuable constant signals on HDL level is not only desirable for decreasing the attack surface for reverse engineers, but also for obscuring the functionality of the design post-synthesis.

Contribution: In this work we present stealthy opaque predicates for hardware, a basic building block to obfuscate constant signals. To this end we discuss criteria for strong opaque predicates, most notably *stealthiness*. Our technique is independent of the hardware platform (Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA)), requires minimal effort by the designer, and is applicably to the majority of all hardware designs. We present two implementation strategies, of which one introduces zero additional gates for the instantiation and no additional latency. Furthermore, we evaluate our technique in two malicious case studies, where we obfuscate backdoors in two cryptographic designs on an FPGA, and demonstrate minimal increases in Look-up table (LUT) and Flip Flop (FF) usage without impacting latency. Finally, we explain how our opaque predicates can be used to mitigate a known removal attack on a watermarking scheme.

Outline: This work is structured as follows: Section 2 provides information on opaque predicates and gives a brief overview on related work in software research. We also define requirements for our opaque predicates and explain our adversarial model. In Section 3 we explain the general idea behind our opaque predicates, before we present two implementation strategies in Section 4. A practical evaluation of the impact on area and latency of our implementation strategies is given in Section 4.3. In the end, we show how our opaque predicates can be used to defend a watermarking scheme from a known removal attack in Section 5 before concluding our work in Section 6.

2 Background

2.1 Static and Dynamic Analysis

When analyzing a product, two general strategies exist, namely (1) static analysis and (2) dynamic analysis. While static analysis investigates the product without any interaction, dynamic analysis is performed while the product is operating. In software reverse engineering, static analysis focuses on the binary or source code, while the code is executed during dynamic analysis. For hardware reverse engineering, static analysis processes images of a delayered chip, a netlist, or HDL description, while the hardware is in addition also simulated or actually run in dynamic analysis. The terms are sometimes used interchangeably with offline- and online-analysis.

2.2 Opaque Predicates

Software research has dedicated a lot of effort to opaque predicates. However, the concept of opaque predicates in hardware has barely been approached yet. In general, an opaque predicate is a function which has a constant Boolean output, regardless of the input. This behavior and the constant output is known to the designer, but non-obvious to an observer. If such a function is found as an expression for a condition in code, static analysis tools cannot be sure which branch is taken. An example for a simple opaque predicate is the equation $x(x+1) \stackrel{?}{=} 0 \bmod 2$ which resolves to *true* for all possible x. Thus, in software, opaque predicates are primarily used for control flow obfuscation. In hardware, opaque predicates can be used, for example, to obfuscate control signals, constant values, or connections to GND or VCC.

Collberg et al. published multiple works on software obfuscation using opaque predicates [CTL03, CT02, CT99, CTL98]. Especially their work from 1998 [CTL98] laid the foundation to formalize and evaluate the effectiveness of software opaque predicates. In the aforementioned paper, the authors present a Java code obfuscator and measure deobfuscation resilience of the generated output. The main purpose of their obfuscator was to harden control flow analysis by inserting suitable opaque predicates into if statements and loop conditions. In other words, it tried to make a control flow graph as obscure as possible. To this end, they presented a variety of techniques to instantiate opaque predicates based on mathematical expressions, as in the example above, pointer equivalence, and concurrency.

Collberg et al. stated that the quality of obfuscation can be described by four criteria: the amount of obscurity added to the program (*potency*), the difficulty to break for an automatic deobfuscator (*resilience*), the computational overhead added by the obfuscation (*cost*) and how well the obfuscation blends in with the existing code (*stealthiness*). We will stick to this terminology when evaluating our own approach. However, these terms describe evaluation metrics for software obfuscation. In the following, we briefly explain how these terms translate to hardware designs.

Potency. The meaning of *potency* transforms directly to hardware designs. It indicates how incomprehensible a design becomes, regarding the target(s) of the obfuscation technique.

Cost. In software, computational overhead describes longer execution time induced by additional lines of code. This transfers only partially to hardware, because computations are often done in parallel. Thus, changes to a design may have no influence on the latency of a design. In addition, area is typically a more crucial cost factor in hardware designs than code size in software. Therefore, cost in hardware measures the overhead in area and latency.

Resilience.   According to Collberg et al., stealthiness is the property of how well an obfuscation blends in with existing code to the "human eye", i.e. *manual* inspection, and resilience measures the difficulty of *automated* deobfuscation. In software analysis, powerful tools exist which automate a majority of the reverse engineering process, hence their output is a suitable indication for resilience. Dedicated algorithms and intermediate results of the automated tools, for example control flow graphs, give a lot of information for manual inspection to the engineer, making stealthiness assessable. Therefore, a separation of stealth and resilience makes sense in software.

Regarding hardware, to the best of our knowledge, no publicly available netlist analysis tool with focus on reverse engineering exists. In 2018, Fyrbiak et al. presented HAL, an interactive graph-based netlist analysis tool which enables partial automation of the reversing process and supports the reverse engineers via a dynamic plugin system [FWS+18]. However, HAL is not yet available as an open-source project. Thus, the majority of the hardware reversing process is still performed manually. Automated deobfuscation can only take place after the manual reversing process revealed the obfuscated parts. Hence, with resilience in hardware we address the difficulty of automated deobfuscation *after* identification.

Note that once an opaque predicate is detected, it can be further analyzed in order to disclose the constant output. Since this step is isolated from the remaining hardware, we expect the complexity, i.e. the resilience, to be low in the general case, especially when aided by tools like HAL. Therefore, we prioritize stealthiness when evaluating our opaque predicates.

Stealthiness.   As mentioned in the previous paragraph, netlist reverse engineering is mainly performed manually, which in turn makes quantifying stealthiness in hardware even harder. The *human factor*, i.e. the individual approach of each engineer, is an **unquantifiable** variable that determines processing speed, reversing skill level, and success rate. In psychological research, first studies were conducted to examine how human beings engage in reverse engineering. Lee et al. had participants deduce the functionality of miniature circuits, for example the functionality of an AND-gate, from a gamified setup with a light bulb and switches [LJL13]. While such studies are an important first step, results are currently too limited to provide quantifiable insights on the human factor in reverse engineering. Therefore, while it is reasonable to demand high stealthiness from strong opaque predicates, we can only argue the existence of this property for a technique, but we cannot prove it.

Software programmers have a large degree of freedom when writing code. Even a bunch of additional instructions does not raise suspicion as long as it appears like other code in the project. Especially instructions related to function calls, math, or Boolean algebra look just like genuine instructions. In hardware, due to the focus on saving area, extensions to a design are quickly discovered if they lead to a strong deviation from expected size, i.e. there is a correlation with the cost metric. Since the main functionalities of a design are usually known to the reverse engineer, he also has an idea about the included building blocks. Additional logical modules in the design can raise suspicion and might lead to identification of obfuscated parts. Thus, in hardware, stealthiness describes blending not only with genuine circuitry (i.e. code in software), but also with logical modules.

Collberg et al. also highlighted the context-sensitivity of this metric, as an obfuscation which is regarded stealthy in one project can be highly unstealthy in other projects. This directly transfers to hardware.

2.3   Previous Work on Opaque Predicates in Hardware

In 2011 Meyer-Bäse et al. implemented the predicates of Collberg et al. in Verilog and VHDL, but they did not introduce new constructions [MBCB+11]. While many publications

exist that target opaque predicates in software, we are only aware of a single work regarding dedicated hardware opaque predicates. It was presented in 2014 by Sergeichik et al. [SI14]. Their approach was to include additional free-running Linear Feedback Shift Registers (LFSRs) into the design. An LFSR always has a Hamming weight larger than zero, regardless of its current state. Thus, by feeding all bits of the LFSR into an `OR` gate, the authors generated a constant 1 signal (or 0 signal by using a `NOR` gate), while the internal LFSR state was constantly changing. However, the technique is attackable because it violates the aforementioned requirement of high stealthiness regarding the logical component. LFSRs where all state bits are OR-ed together blend in well with existing circuitry since they are not introducing major overhead in area. Nevertheless, this logical structure is highly uncommon in hardware designs as it has no other purpose than providing a constant signal. Exploiting this property, Wallat et al. presented a practical removal attack on the opaque predicates by Sergeichik et al. in 2017 [WFSP17]. They searched for the special LFSR structure in a netlist and replaced it with a connection to GND or VCC, in a fully automated fashion.

2.4   Threat Model

We assume the same threat model as prior research on hardware analysis [AK07, CB09, RKK14, BHBN14, STBF17]. The adversary, i.e. reverse engineer, has full access to a flattened gate-level netlist. He knows the general functionality and is aware of the input and output specifications, but has no knowledge on internal workings. His goal is to (1) identify the obfuscated element, (2) analyze, and (3) deobfuscate it. We emphasize that we especially focus on adversaries which employ static analysis to reverse a design, as this is what opaque predicates are designed to protect against. However, in Section 3.1 we argue that our construction even remains stealthy to some extend when faced with dynamic analysis.

Regardless of the employed analysis strategy, an adversary breaks an obfuscation if he succeeds in the third step, i.e. he is able to look through the obscurity added by the obfuscation. For example, he may want to replace an obfuscated module with a plain equivalent or extract implementation details from an obfuscated design. To this end, he has to (partially) reverse engineer relevant logical modules in both control and data path. Note that an "adversary" not necessarily has to be malicious. A designer who tries to detect obfuscated backdoors is also included in our definition.

As demonstrated for numerous settings, it is realistic to assume that the adversary has access to a flattened gate-level netlist: It can be extracted at chip-level or via layout reverse engineering [QCF+16, VPH+17] in the case of ASICs, via bitstream-level reverse engineering [SFK+16, PHK17] in the case of FPGAs, or is obtained directly from the IP provider [RKK14].

We emphasize that - given infinite resources - reverse engineering always succeeds, simply because obfuscation must not strip functionality from the design. This is also formalized in the impossibility result for perfect obfuscation by Barak et al. [BGI+01]. However, the cost in both time and effort can quickly grow to extents where reverse engineering is regarded as practically infeasible. Due to the amount of unknown variables in a design, i.e. the number of gates and the lack of in-depth knowledge on design specifics, we assume that reverse engineering of the full design is out of scope. In fact, any obfuscation technique would be broken in this case.

2.5   Requirements

Our technique is applicable to the majority of all designs. The only requirement is that it has to include a counter or Finite State Machine (FSM). We will make use of already existing modules in order to minimize costs. With more instances of FSMs or counters in

the design stealthiness and resilience of our approach increases. We regard this requirement as rather weak, since most IP-cores contain such building blocks with the exception of, for example, fully combinatorial or pipelined designs.

3   Our Construction

In this section we explain the high-level idea behind our opaque predicates before we give detailed implementation strategies in Section 4.

The goal of our construction is to substitute constant signals with *obfuscated constants* which are difficult to detect via static analysis: An obfuscated constant may change its value arbitrarily throughout the operation of the device, but has a known and stable value at certain periods in time. In order to achieve high stealthiness, we identify and leverage suitable existing signals or introduce small additional circuitry which we connect to the places where the stable value is needed.

In software, everything which is not affected by the current line of code is static. Hardware, however, runs in parallel, and signals in the design may change all the time. The value of a signal is only evaluated if the data or control port it reaches is active, for example the data port of an enabled FF. Therefore, this dynamic switching behavior does not impact functionality if connected ports are deactivated, which is done by control logic. Consequently, the utilization of obfuscated constants does not change functionality if the stable value is hold as long as succeeding ports are active. We call this period where the original value has to be available to maintain functionality *processing period.*

The main idea to achieve our goal is to use the state registers of FSMs or counters as opaque predicates. Note that these are frequent elements in IP-cores, so their inclusion itself does not raise suspicion in contrast to the construction of Sergeichik et al. [SI14] (c.f. Section 2.3). This also explains our requirement that the original circuit has to include such elements.

FSMs and counters are elements that transition between states based on certain conditions, for example a trigger signal or the previous state. During the time where no condition is met, the register which encodes the state does not change. This lays the foundation of our opaque predicates since there are existing elements that have known values at certain periods of operation. We use such a state register to generate obfuscated constants.

Since FSMs generate sequences of states analogously to counters which generate sequences of numbers, FSMs can be regarded as counters with non-numeric counting styles. For the sake of simplicity we restrict our explanations and examples to FSMs in the following, but emphasize that presented techniques are similarly applicable to any kind of counters.

Since the sequence of states in an FSM cannot be changed without altering device functionality, the designer has to select a suitable FSM to ensure a stable signal during the processing period.

Example   Figure 1 shows a small example. FSM_1 controls FSM_2 which in turn controls a computation. After the computation is done, FSM_1 triggers transmission of the result. This process is repeated indefinitely.

If a constant expression is only required by logic which is active in the *Work1*, *Work2*, and *Work3* states of FSM_2, there are two options to instantiate an opaque predicate: Over the whole processing period, FSM_1 remains in the *Wait* state, i.e. the value of the state register is constant and can be used as an opaque predicate. FSM_2 passes *Work1*, *Work2*, and *Work3*, so all FFs of the state register of FSM_2 which do not change in these states can also be used to generate the constant expression.
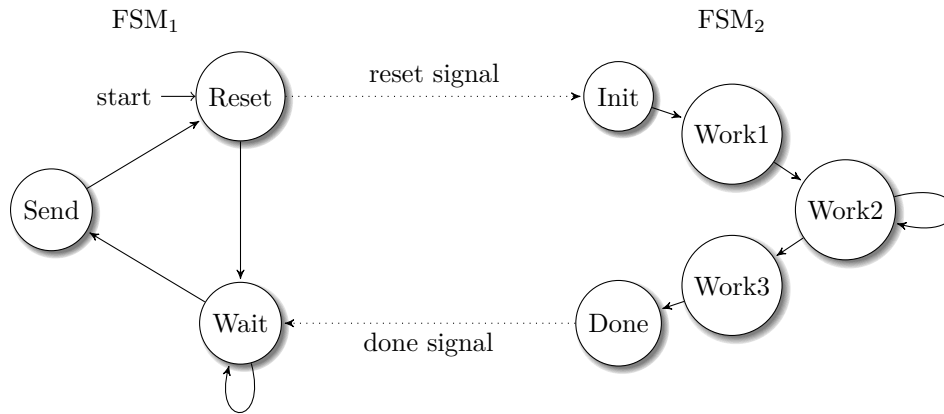
Figure 1: Visualization of two FSMs which can be used as opaque predicates.

3.1 Discussion

We discuss the potential of our opaque predicates according to the criteria originally proposed by Collberg et al. (c.f. Section 2.2). Furthermore we elaborate on the strength of our opaque predicates when faced with dynamic analysis and inspect the applicability of known techniques against software opaque predicates to our approach.

Potency. With respect to potency, after applying our approach the constant signal is entirely gone when simply "reading" the design, i.e. flattened gate-level netlist. Therefore, regarding the goal of obfuscating constant signals, we introduce maximum obscurity.

Cost. In terms of cost, we do not introduce any additional gates on HDL level when using existing FSMs. Depending on the actual instantiation, additional wires are required. However, as we show in our case studies in Section 4.3, the synthesis toolchain will add a small number of additional gates, for signal routing and since previous optimization is not possible anymore. For example, an `AND` gate where one input is constant can be optimized by the compiler, but as soon as both inputs seem variable, no optimization can be performed and the gate has to be instantiated as is.

Stealthiness. As we argued in Section 2.2, stealthiness cannot be quantified easily. Therefore, the best we can do is to present arguments why our opaque predicates are stealthy. Since we utilize existing circuitry, the approach is stealthy in a sense that minimal additional gates and no new FFs are introduced. Intuitively, this leads to no unusual structures in contrast to the approach by Sergeichik et al. (c.f. Section 2.3). Our building blocks, FSMs and counters, are common in IP-cores, so their existence is not suspicious either. Since an FSM controls logical modules of a core depending on its current state, signals that reach from a state register into the design are unobtrusive. Even signals from state registers which are connected to the data inputs of FFs are common in circuits, as can be seen in our two case studies, which we present in detail in Section 4.3. We counted the FF instances, where the combinatorial circuit which generated the FF's data input had at least one FSM signal as input in relation to all FFs which receive any signal generated via an FSM. For our (small) PRESENT case study we identified 28.79% of all FFs, for our larger RSA case study we identified 37.62%. Thus, a reverse engineer cannot use this property to immediately identify obfuscation instances.

Therefore, if an adversary is looking for opaque predicates in general, we are confident that our technique will go unnoticed. Furthermore, by generating a signal from multiple

opaque predicates, i.e. the state registers of multiple FSMs, the original functionality becomes obstructed even more. In that case, multiple individual components, which are not related in the first place, contribute to the signal generation, thus requiring a reverse engineer to analyze an even bigger part of the circuit.

However, even if an adversary is looking especially for our opaque predicates we argue that this is only successful with tremendous efforts. We stress that, while efficient algorithms exist to extract FSMs from a gate-level netlist as shown by Shi et al. [S+10], this step is not the major difficulty in detecting our opaque predicates. As mentioned before, FSMs naturally control other circuitry. These control signals are outputs of the state register. The adversary would have to reverse engineer the whole datapath of the logical modules which are controlled by the analyzed FSM to detect that a processed signal is actually constant. To this end, he would also have to deduce the processing period. We stress that all this additional information, which is required to successfully detect our opaque predicates, tremendously increases the required reversing effort.

Several approaches exist to make sense of a circuits datapath. In 2012, Shi et al. presented an automated technique to extract functional modules from a circuit [S+12], by first stripping all FSMs from it and subsequently matching subcircuits against abstract models. At the same time Li et al. developed a technique based on behavioral patterns to generate a high-level description of unknown circuits [L+12]. Note that this technique uses dynamic analysis. One year later, Li et al. presented another technique to automatically detect word-level [L+13]. Extending on this technique, Subramanyan et al. automated extraction of registers and adders [S+13, S+14] and Gascón et al. improved bit-order reversing using template matching [G+14]. However, all of these techniques only help in recovering modules of the datapath. Making sense of interconnections and, regarding our construction, identifying processing periods, still has to be performed manually and in conjunction with the control logic. Consequently, we regard our opaque predicates as stealthy.

Resilience. As noted in Section 2.2, resilience in hardware becomes important once an obfuscation has been identified, i.e. after stealthiness failed. If an adversary manages to confidently identify an instance of our opaque predicates including the processing period, deobfuscation is straightforward. By simulating the FSM in isolation, i.e. applying a very restricted dynamic analysis, the constant value during the processing period can be disclosed. Therefore, our approach features low resilience.

Dynamic Analysis. As noted in Section 2.2 and Section 2.4, opaque predicates are designed to defend against static analysis. However, we want to shortly argue why our construction also offers potential when faced with dynamic analysis. Since dynamic analysis allows for inspection during design operation via simulation or execution, an adversary would be able to detect signals which do not switch in certain periods of time. However, this is also (partially) normal behavior of control signals. Since the FSMs which are used as opaque predicates also control other parts of the circuit, their signals naturally remain constant for certain periods of time, for example to enable parts of a circuit or to act as state-dependent bitmasks. Hence, while identifying gates which receive a constant input over a certain duration is possible with dynamic analysis, they do not necessarily have to be the target of an opaque predicate. Therefore, dynamic analysis is only capable of narrowing the parts of the datapath which have to be analyzed to those including the identified FFs, but does not instantly break our construction.

Techniques Against Software Opaque Predicates. Multiple approaches exist to attack software opaque predicates. For example, Dalla et al. presented an attack based on dynamic analysis to identify software opaque predicates of the form $f(x) = 0 \bmod y$ [DPMDBG06].

They observe conditional jumps over a number of executions and propose a technique based on abstract interpretation to detect opaque predicates in these conditions. Ming et al. presented LOOP, a *L*ogic-*O*riented *O*paque *P*redicate detection tool using static analysis after a symbolic execution [MXWW15]. To this end, they examine all conditions on the execution path, both in isolation as well as in conjunction to detect correlated predicates.

In general, approaches that target software opaque predicates all share a common structure. First, they identify branch conditions and then further evaluate/compare satisfiability of these conditions in order to identify opaque predicates. This is a well suited approach in software since opaque predicates are used for control flow obfuscation and do not impact other computations.

We stress that this general approach cannot be plainly transferred to hardware circuits. In software, control flow is observable from conditional branches, but data flow is obscure by the nature of pointers, while in hardware data flow is observable from wires and combinatorial logic between FFs, but control flow is obscure because of state machines. Note that understanding the data flow is distinct from understanding the meaning of the datapath in hardware, it only describes **where** information goes, not what it means. Our construction, reuses existing signals which are also used at multiple other places in the circuit, i.e. there is no isolated use for obfuscation. Therefore, from our point of view, techniques against software opaque predicates cannot be applied to our opaque predicates.

4   Implementation Strategies

In this section we present two implementation strategies. The first one explains in detail how an existing state machine can be used as an opaque predicate. The second strategy can be used if no existing state machine or counter is suitable for the first method.

4.1   Implementation in Existing Circuitry

In order to use an existing FSM as an opaque predicate the designer has to analyze which of the available state registers are suitable. He first has to identify the processing period T, consisting of starting event and duration of the constant to obfuscate. He then has to verify that the FSM passes the same set of states S during each iteration of T. If S changes for several occurrences of T, the respective FSM cannot be used as an opaque predicate. This can be the case, for example, if the FSM is independent of the circuitry which processes the constant. If the set of passed states is stable for each iteration of T, the state machine is a valid candidate.

After selecting a valid candidate the designer has to configure the state encoding of the FSM. The encodings of all states not in S can be chosen arbitrarily. For all states in S, encodings have to be selected where at least one common bit remains constant. Consequently, the outputs of the corresponding FFs remain constant during T and can be used to generate the signals of an obfuscated constant.

Using this implementation strategy no additional FFs and no combinatorial gates are added to the design. However, as becomes evident in the case studies in Section 4.3 additional gates may be inferred due to optimization issues.

Example.   An FSM with a 5-bit state register $(\mathrm{FF}_4, \ldots, \mathrm{FF}_0)$ is used as an opaque predicate to generate the 7-bit constant $(C_6, \ldots, C_0) = \mathtt{1101000}_2$. The designer identified that during period T where the constant is required the FSM consistently passes three states that form the set S. He encodes them as follows: $S = (\mathtt{10100}_2, \mathtt{11000}_2, \mathtt{11100}_2)$. Therefore, he can use FF_4, FF_1, and FF_0 which constantly output 1, 0 and 0 during T to generate the obfuscated constant. In the remaining states of the FSM those bits are not

fixed and therefore the output appears to have dynamic behavior. A possible instantiation would be to connect these FFs as shown in Figure 2.
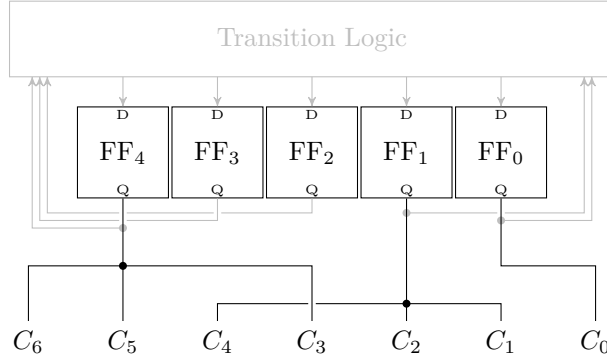

Figure 2: Exemplary constant generation from a state register.

4.2   Implementation with Additional Circuitry

If the designer is in a situation where none of the existing FSMs or counters offer a stable set S during the processing period, he can still make use of our general idea. This situation can for example occur if the design consists of independent FSMs where the state sequence during the processing period changes depending on external events. In this section we describe an implementation strategy which adds a small number of gates and FFs in order to create a module that looks like a genuine FSM. However, the new FSM will reach a known stable state after a number of clock cycles, thus providing constant values after the initial delay. The designer only has to add the FSM to the design and connect the reset signal to be triggered at a point which is far enough ahead of the processing period. Note that generating an FSM which cycles a known sequence of states during the processing period is highly difficult in this scenario. In the best case, we are only able to give a lower bound for the time difference between reset and processing period. By making sure that the additional FSM is in its stable state simply before the processing period is reached, we do not have to care for varying amounts of time after FSM reset.

The difficulty in generating this opaque predicate is that we have to find interconnections for the state register's FFs which result in a stabilization from a known starting value, i.e. the reset value, to a known stable value after a known number of clock cycles. We present two algorithmic approaches which lead to three algorithms QM, QMX, and RND that automatically generate such interconnections. They require the register size n and start/reset value s as input as well as a stabilization delay t. The algorithms output the stable state z and the corresponding function $f : \{0,1\}^n \rightarrow \{0,1\}^n$ which represents the interconnections. Note that all algorithms can be adjusted to regard the input t as a lower bound t_{min} and internally select a random $t \geq t_{min}$. In that case, the actual stabilization delay t becomes an additional output.

The transition function of the FSM during each clock cycle is then described by the function $F : \{0,1\}^n \times \{0,1\} \rightarrow \{0,1\}^n$ where

$$F(x, RST) = \begin{cases} s & RST = 1 \\ f(x) & else \end{cases}$$

i.e. one clock cycle in the hardware implementation updates the register from state x to state $f(x)$ or resets the register if required. In the following we first present the three algorithms before comparing their outputs.

Approach 1: Quine-McCluskey Supported Generation.   Our first approach chooses a random sequence of states and utilizes Boolean algebra to generate an according transition function f. The chosen sequence $X = (x_i)_{0 \leq i \leq t+1}$ is of length $t + 2$, where x_0 is set to s and $x_1, \ldots, x_{t-1}$ are chosen randomly such that no state appears twice. The two final states in the sequence have to be equal, i.e. $x_t = x_{t+1}$. When f is generated based on X, it outputs said sequence, thus $f(x_i) = x_{i+1}$ holds and subsequently $f(x_t) = x_{t+1} = x_t$, i.e. the function stabilizes after t clock cycles in state $z = x_t$.

We use the Quine-McCluskey algorithm to generate minimized Boolean functions for each bit separately [Qui52]. The Quine-McCluskey algorithm basically transforms a truth table into the corresponding Boolean function. The truth table can also contain so called "don't care states" where the output is irrelevant for the respective input values. The algorithm requires a set of values which shall result in a 1 at the output and the "don't care states". It then returns a Boolean function in Disjunctive Normal Form (DNF) which outputs a 1 for all specified inputs, either 1 or 0 for all "don't care" values and a 0 for all other inputs. In our case, all states which are not in the sequence are "don't care states".

The original Quine-McCluskey algorithm outputs a function in DNF, i.e. it only uses `AND`, `OR`, and `NOT` gates. However, modifications exist which also make use of `XOR` gates. In the following, we refer to the above algorithm that uses the original Quine-McCluskey as `QM` and to the one using the XOR-enhanced variant as `QMX`.

Approach 2: Randomized Generation.   In contrast to our first approach, the second one generates a random transition function f for an n-bit state register and then simulates the sequence of generated states in order to determine suitability. The algorithm, further denoted as `RND`, also processes each bit individually when generating f. For each FF in the register, it first selects a random subset I of all other FFs of said register, where $|I| \geq 2$ has to be ensured. The output signals of all FFs in I are put through an inverter stage where a randomly selected subset is negated. The output signals of the inverter stage are then passed as inputs to an `OR`, `AND`, or `XOR` gate, which is again randomly selected. The output of that gate is the output bit of f, which is used to update the currently processed FF. An exemplary interconnection for a single FF of a 5-bit state register $(\text{FF}_4, \ldots, \text{FF}_0)$ as generated by this algorithm is shown in Figure 3. In this case $I = \{\text{FF}_0, \text{FF}_2, \text{FF}_3\}$ was chosen, the output of FF_0 was selected to be the input of a `NOT` gate in the inverter stage, and an `OR` gate was instantiated to finally combine the signals.
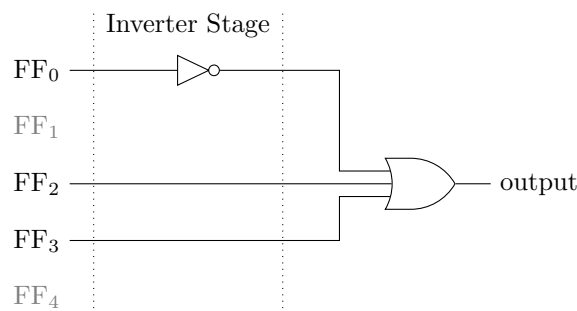

Figure 3: Exemplary interconnection as generated by the `RND` algorithm.

After generating f as described above, simulation starts at state s and simulates until a stable state z is found. If the number of clock cycles until f stabilizes in state z is $\geq t$, the algorithm outputs f and z. Otherwise a new function f is chosen and the process is repeated.

Table 1: Comparison of additional gates required for the interconnections of state register FFs.

	n	3			4				5			
	t	2	3	5	2	3	5	10	2	3	5	10
QM	#not	2.6	2.9	3.0	3.5	3.8	4.0	4.0	4.3	4.7	5.0	5.0
	#and	3.8	4.3	6.1	6.7	8.3	12.8	24.6	10.0	13.6	22.0	45.7
	#or	5.1	5.4	5.3	9.5	11.3	13.7	16.0	14.6	19.0	25.2	34.5
	#xor	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QMX	#not	1.9	2.4	2.9	2.3	3.0	3.9	5.4	2.3	3.3	4.5	6.9
	#and	0.7	1.4	3.0	0.8	1.8	4.8	13.5	0.9	2.2	6.6	20.9
	#or	5.1	5.6	5.9	9.1	11.3	15.0	21.6	13.4	18.3	26.5	43.7
	#xor	1.8	1.9	2.3	3.2	4.0	5.8	9.0	4.9	6.8	10.8	19.0
RND	#not	2.4	2.5	-	3.3	3.3	3.2	3.0	4.3	4.2	4.2	4.1
	#and	1.3	1.2	-	2.3	2.2	1.8	0.8	3.7	3.5	3.0	1.8
	#or	1.7	1.8	-	3.6	3.6	3.8	4.1	5.8	6.0	6.2	6.8
	#xor	0.5	0.5	-	1.3	1.5	1.9	3.3	2.3	2.6	3.3	4.9

Discussion. Our algorithms tackle the same task from different directions. While QM and QMX take a bottom-up approach of selecting a state sequence and generating the corresponding transition function, RND takes the top-down approach of selecting a transition function and verifying its suitability.

Since opaque predicates are generated during the design process, we regard runtime as an insignificant property of our algorithms. For the sake of completeness we shortly elaborate on the respective expected execution times. For a truth table with k entries Quine-McCluskey terminates after $\mathcal{O}(3^k \ln(k))$ steps (worst-case), therefore QM and QMX terminate in a similar amount of steps. Since RND solves the same problem, it terminates as well, but accurate runtime estimations are difficult because of its random nature. According to our experiments, if t gets closer to 2^n, the maximum number of states an n-bit register can encode, the random search can become very long.

All three algorithms have to add additional circuitry, namely n FFs and the gates required to realize f. Table 1 shows the amount of gates each algorithm requires to generate f, averaged over 1000 executions for each parameter set. Since f is generated independently for each FF, there may be common terms in the Boolean functions. While a compiler could optimize this during synthesis, we employed a postprocessing step directly on f to extract common terms which have to only be computed once in order to get a fair comparison in Table 1. We explain the parameter choices later when we extend on stealthiness, as both topics are related.

For the RND algorithm we did not compute values for $n = 3, t = 5$. With $n = 3$ only 8 states are possible and randomly finding f that stabilize after ≥ 5 clock cycles took minutes to hours, while results of the first hundred iterations indicated no different behavior compared to the other inputs. It is easy to see that RND introduces fewer additional gates than the output of the Quine-McCluskey supported algorithms. Furthermore, the amount of gates required for RND is mainly influenced by the choice of n while for QM and QMX n and t both notably impact the gate count. The advantage of RND becomes more significant for bigger values of n and t.

All algorithms can be tweaked to use different strategies for handling the input t, as for example it could be interpreted as a lower bound t_{min} for additional randomness. However, RND offers additional degrees of freedom for the actual implementation: When generating a random f, the strategy of creating connections can be adapted, e.g. more or less inputs can be considered for each bit or multiple gates can be inserted. The output of the Quine-McCluskey method is always in DNF, a structure where numerous AND gates

are input to `OR` gates. An FSM where the state transition logic is in DNF structure could offer an anchor point for detection to a reverse engineer.

This leads to our conclusion that the randomized generation returns much smaller and thus more natural, i.e. more stealthy, interconnections and can be customized to a larger extend than the Quine-McCluskey supported generation. Therefore, we regard the randomized approach as the algorithm of choice.

We stress that regardless of the employed algorithm, there is a stabilization delay. Depending on the point in time where the reset signal is triggered, the designer might have to infer an artificial delay to ensure that the FSM stabilized before the processing period.

Stealthiness and Parameter Choice. Since we add new circuitry to the existing design with this approach, stealthiness is lower than for the first variant. Still the inclusion of an additional FSM in a design with multiple benign FSMs is not suspicious in the first place. In order to integrate the FSM into the whole design and not make it appear as an outsider, it should be reset by other state machines. That way it gets integrated into the control hierarchy and blends in better than an autonomous module. To further improve similarities with existing FSMs, parameters n and t should be selected to output FSMs which do not vastly differ from other FSMs in the design in terms of size. For example could an FSM with a 15-bit state register be suspicious in a design where each module only has a low amount of logical states. Because of this Table 1 was created with $n \in \{3, 4, 5\}$ as these were common FSM register sizes we experienced.

Stealthiness is only barely affected by t. Even for small values, the generated interconnections are not suspicious during static analysis. Only during simulation, i.e. dynamic analysis, larger values mean that the stabilizing behavior will occur later. Therefore, the designer can select a value of t which allows for easy integration. To map this property we selected arbitrary values in Table 1 to only analyze the impact on additional gates.

Static analysis, presented with a gate-level netlist, cannot determine from the wiring that the FSM produces a stable signal after a number of clock cycles from reset. Only via simulation this behavior becomes evident. However, even if this property is uncovered, the opaque predicate is not immediately identified as such. FSMs often control a sequence of operations before idling in a *Done* state, i.e. even genuine FSMs may stabilize in a certain state. In order to actually identify our opaque predicate FSMs, still further reverse engineering of the influenced datapath is required, immensely increasing the attacker's efforts.

The technique with additional circuitry introduces some potential weaknesses. The new FSM is mostly self-contained, i.e. only the reset is performed by external control signals. While such FSMs *can* be a normal part of some circuits, they might be a suspicious module in others, resulting in a comparable weakness to the approach of Sergeichik et al. [SI14] (c.f. Section 2.3). However, without further inspection our modules still look just like genuine FSMs in the general case and require an adversary to invest additional effort to detect.

We therefore conclude that our opaque predicates with additional circuitry still remain stealthy, although they are weaker than our opaque predicates in existing circuitry.

4.3 Practical Evaluation

In order to evaluate the costs of our opaque predicates, we observe the change in the number of LUTs and FFs when applying our two implementation strategies to FPGA designs. To this end we implemented two case studies where we obfuscate Trojans in cryptographic cores with our techniques to harden against detection via reverse engineering. Designers are afraid of purposely implemented backdoors in designs since the crypto wars in the 1990s and this fear has been rekindled by the Snowden revelations in 2013. Hence we regard this scenario as a realistic target for obfuscation.

The first case study targets a PRESENT encryption core with input triggered Trojan and the second case study an RSA core, subverted with a kleptographic backdoor. Although we executed our case studies on FPGAs, our technique is in no way exclusive to this architecture but can also be used on ASICs.

Our expectations are that for the instantiation in existing circuitry, we observe no change in the number of LUTs and FFs while for the approach with additional circuitry, the amount of FFs increases by n and the amount of LUTs slightly increases because of the feedback logic and routing.

In the following we briefly introduce the designs before presenting the measurement results.

4.3.1   Trojanized PRESENT

The first case study features a trojanized PRESENT encryption core as illustrated in Figure 4. The general Trojan idea is similar to the design in [SFK+16]. When active, the Trojan replaces the user-supplied key with a fixed embedded key. To an observer, the output still looks encrypted, and multiple infected devices still successfully communicate. The scenario that multiple devices are infected is likely, if the Trojan is included in the IP-core that was used when designing the device family.

Thus, with knowledge of the trigger and the fixed key, an attacker can decrypt and manipulate any traffic. The Trojan is input-toggled, i.e. once the trigger plaintext is received the Trojan stays active until it is presented a second time with the trigger plaintext. Note that this Trojan can be used by anyone who has knowledge of the trigger value and the fixed key, so the attacker does not have to be the Trojan designer. We use opaque predicates to obfuscate the fixed key and the trigger value. We emphasize, that this protects only the constant key and trigger value, not the Trojan or its trigger mechanism itself, against detection. For example, Trojan detection techniques such as MERO by Chakraborty et al. [CWP+09] still apply. Via statistical extraction of rarely active paths and automatic test pattern generation (ATPG) the goal of MERO is to generate a minimal test set which stimulates each of these rare paths multiple times. During dynamic analysis, while feeding the inputs of that test set, triggering a Trojan is highly probable. Regardless of our opaque predicates, this approach would still detect our Trojan. Intuitively, an obfuscation technique which transforms rarely active paths to paths with high activity could potentially defend against detection via tools like MERO. We leave this as an open question for further research.
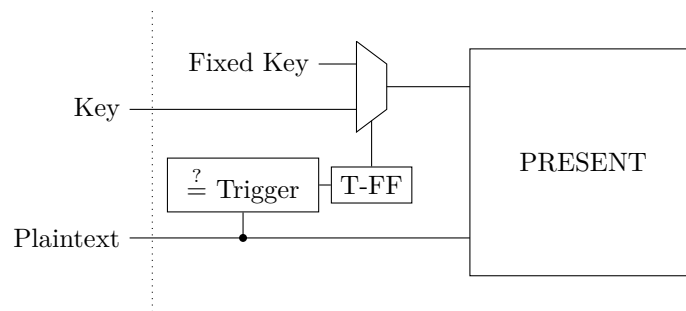

Figure 4: Visualization of the Trojanized PRESENT core.

Obfuscation Impact.   Since we protected the constant values in the design, we focus on an reverse engineer who tries to extract the trigger value and the fixed key or performs a quick scan for information which hints unexpected constant values in the original HDL code. For example, in order to find the trigger value, the reverse engineer could scan for

comparators with fixed values. For the fixed key, he could search for an isolated control signal which is spread to all data inputs of a register of the same size as the key. Our opaque predicates render any kind of attack ineffective which focuses on information introduced by constants in HDL code. For example, the obfuscation forces the synthesis toolchain to turn the comparator with a fixed value into a dynamic comparator, removing any indication about a constant trigger value. The adversary could still find a comparator, but to make sure that this is a fixed-input trigger or to extract the trigger value, further reversing is required.

Implementation. We implemented the Trojan three times, once unobfuscated, i.e. as depicted in 4, once using an existing FSM and once applying our strategy with additional circuitry, generated via the `RND` algorithm. A simple UART core was added to handle I/O. For the approach with additional circuitry, we generated an opaque predicate of size 3 bits as this was the average size of FSM state registers in the design. We synthesized, placed and routed the design for a Xilinx Artix 7 35T FPGA. The optimization strategy was set to optimize for area and the design was fully flattened.

4.3.2 Subverted RSA

The second case study features an RSA core where key generation is subverted with a kleptographic backdoor. Kleptography has been introduced by Young and Yung around 1996 [YY96, YY97]. In recent years there has been numerous research which presents kleptographic Trojans on cryptosystems [YY07, GGKL08, GKK$^+$06, KLT17]. Kleptography describes the study of subliminally and securely stealing information. The output of a device with a kleptographic Trojan remains conform to specifications when observed with black-box access, but leaks sensitive information exclusively to the Trojan designer. To this end, it asymmetrically encrypts (parts of) the sensitive information with an embedded public key of the designer, leaking this so called "kleptogram" over one or multiple outputs. In this case study we use our opaque predicates to obfuscate such an embedded key.

We implemented our own kleptographic Trojan which replaces the key generation of the RSA parameters (n, e, d) as shown in Algorithm 1. It uses the Trojan designer's RSA public key (N_{adv}, E_{adv}) to generate kleptograms. The advantage of this approach is that kleptograms are generated without requiring any additional hardware modules. Only small changes to the datapath and a single FSM were enough to subvert the original core. This is a difference to existing kleptographic attacks on RSA, where additional modules, for example a dedicated hash function, were required. However, our kleptographic Trojan only works on RSA key generation where e is not fixed but randomly chosen.

Algorithm 1 Subverted RSA KeyGen

  Input: 1^λ
Output: pk = (n, e), sk = (d)
 1: Choose p, q as random $\lambda/2$-bit primes
 2: $n \leftarrow pq$
 3: $e \leftarrow p^{E_{adv}} \mod N_{adv}$
 4: **while** $\gcd(e, \Phi(n)) \neq 1$ **do**
 5:     $e \leftarrow e + 1$
 6: $d \leftarrow e^{-1} \mod \Phi(n)$
 7: **return** pk = (n, e), sk = (d)

Instead of randomly choosing the public exponent e, it is computed as $e = p^{E_{adv}} \mod N_{adv}$ and then incremented until it is invertible modulo $\Phi(n)$. Subsequently, the Trojan designer has to compute $p' = (e - i)^{D_{adv}} \mod N_{adv}$ for a small number of i, until

n is divisible by p'. Then he can factorize n as $p = p'$, $q = n/p$, break RSA, and thus decrypt or even manipulate all ongoing communication.

Obfuscation Impact.   Note that in contrast to the PRESENT case study, our kleptographic Trojan is always active. Techniques for Trojan detection often aim at finding triggers or circuitry which is inactive most of the time (c.f. [CWP+09, WSS13, ZYW+13]) and are therefore not capable of detecting kleptographic Trojans.

We try to protect against a reverse engineer who tries to detect a kleptographic Trojan by using the knowledge that the design has to include the Trojan designer's public key. As the key pair (N_{adv}, E_{adv}) has to be included in the design, he could try to locate constant signals which form a possible input to the exponentiation engine. We use an opaque predicate to obfuscate N_{adv} and E_{adv} in order to harden against this detection approach, as all inputs to the exponentiation engine now appear to be dynamic without deep inspection.

Implementation.   We selected an exemplary 256-bit RSA-core, capable of encryption, decryption, and key generation. Again, we implemented the design three times: unobfuscated and once with each of our two techniques, using the `RND` algorithm for our approach with additional circuitry. We use the same UART core as in the previous case study to handle I/O. When applying our first implementation technique we used a single FSM to generate both values N_{adv} and E_{adv}. For the approach with additional circuitry, we generated an opaque predicate of size 5 bits as this was the average size of FSM state registers in the design. We synthesized, placed and routed the design for a Xilinx Artix 7 35T FPGA, to get comparable results to the first case study. The optimization strategy was set to optimize for area and the design was fully flattened.

4.3.3   Implementation Results

Table 2: Area comparison of our three case studies. Percentual changes refer to the unobfuscated circuit.

Design		LUTs		FFs	
PRESENT	Unobfuscated	304		347	
	Strategy 1	307	+0.99%	347	+0%
	Strategy 2	304	+0%	350	+0.86%
RSA	Unobfuscated	10570		5316	
	Strategy 1	10811	+2.28%	5314	−0.04%
	Strategy 2	10692	+1.15%	5323	+0.13%

Table 2 shows the results of our evaluation in terms of LUT and FF count. It contains the absolute numbers for each variant of the two case studies as well as the relative increase compared to the unobfuscated design. *Strategy 1* refers to the implementation which uses an existing FSM, and *Strategy 2* to the one where new circuitry for the opaque predicate is added. We confirmed manually that the additional circuitry remained untouched by the synthesizing toolchain, which is another indicator that they are difficult to detect via static analysis, since compiler optimization is based on the results of a static analysis. For *Strategy 2*, we connected the reset signals of our opaque predicates to reset signals of FSMs which were reset far earlier in both case studies. Therefore, our obfuscations have no impact on latency as the opaque predicates had time to stabilize before the processing period. Although *Strategy 1* is expected to have no impact on area, as we solely use existing circuitry, the number of LUTs increases by 0.99% for the PRESENT case study and by 2.28% for the RSA core. We expect this increase to result from layout optimization of the synthesis toolchain to fit the FPGA architecture. Regarding the amount of FFs,

our expectations are met, as no additional FFs were instantiated. In the RSA case study, the optimizer managed to take advantage of the new connections and reduced the number of FFs by 0.04% compared to the unobfuscated design.

For *Strategy 2* we expected a slight increase in both, FF and LUT count. For the PRESENT case study there is an increase of exactly 3 FFs, i.e. the FFs which were added for the opaque predicate. However, in the RSA case study, 7 new FFs were added, although the opaque predicate only uses 5. We assume that the additional a FFs are internal buffers, also inferred by the optimization tools. Surprisingly the amount of LUTs did not increase for the PRESENT core. Apparently, the additional logic can be embedded into underutilized LUTs. In the RSA case study the number of LUTs increases by 1.15% which reflects our expectation, although the increase exceeds the required gates for the f function.

Our evaluation shows that the theoretical expectations are not exactly met in reality. Although the impact on the amount of FFs is as expected, the number of LUTs increases more than anticipated. Optimizations applied by the synthesis toolchain heavily influence the numbers. Note that the toolchain itself was not able to deduce that our construction actually generates a constant value and did not replace it. While we did not achieve a zero-overhead obfuscation using our implementation strategies, the results still attest very low overhead.

5   Application to Watermarking

In the previous section we presented two case studies which focused on malicious application of obfuscation techniques to hide a Trojan. In this section we present the theory of a defensive application of our opaque predicates, where a designer tries to protect intellectual property.

We apply our obfuscation to the watermarking scheme of Schmid et al. [SZT08], effectively rendering previous removal attacks ineffective. The watermarks of Schmid et al. are used to protect IP-cores on LUT-based FPGAs and can be verified from the bitstream file that is configuring the FPGA. The configuration of a LUT in the bitstream file defines the output values for each input combination. Subsequently, for an n-to-1 bit LUT 2^n bits are required for configuration. The idea of Schmid et al. was to fix some inputs of a LUT to 0, i.e. GND. Therefore, all input combination where those inputs would be 1 cannot occur during operation. The bits in the LUT configuration which are intended to configure the corresponding output bits are then used to encode a watermark. As watermarks may not fit in one LUT configuration, it can be split over multiple LUTs.

An example is illustrated in Figure 5 which shows the truth table and resulting configuration for a 4-to-1 LUT. The inputs I_3 and I_2 are fixed to GND, so only the input combinations for the four bits labeled C can actually occur during operation. We marked all other input combinations which cannot occur gray. For example, the input combination $(I_3, \ldots I_0) = 1000$ cannot occur as I_3 is constantly 0. The 12 configuration bits that belong to these inputs, labeled W, can be used to encode a watermark.

GND $\rightarrow I_3$	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
GND $\rightarrow I_2$	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
I_1	0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
I_0	0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
output/config	C C C C W W W W W W W W W W W W

Figure 5: Example of a 4-to-1 bit LUT configuration where the C bits are actual configuration and the W bits can be used for watermarking.

The bitstream file used to configure the FPGA contains all LUT configurations and thus the watermark's existence can be verified via exhaustive search. Removal of the watermarks on bitstream level is considered infeasible, since no information on interconnection is plainly available. Removal attacks on the netlist level are regarded as far more likely. Therefore, Schmid et al. include an attacker, who has access to the netlist of the design, into their attacker model.

In 2017 Wallat et al. presented a practical removal attack on that scheme [WFSP17]. With access to the netlist and the information that unused LUT inputs are connected to GND, tracing all GND connections to LUTs instantly reveals the watermarked instances. Subsequently removal of the watermark is a straightforward task.

To counter removal attacks, Schmid et al. already proposed the insertion of bogus cells that generate constant signals. Wallat et al. directly propose the use of opaque predicates to render their attack useless but also highlight the lack of hardware opaque predicates.

Using our instantiations one can efficiently harden the watermarks of Schmid et al. against netlist level removal attacks. The idea is to connect the fixed LUT inputs to an obfuscated constant that outputs a zero during the processing period instead of connecting directly to GND. Since this connection does not affect LUT configuration the watermark detection algorithm remains unaltered, but attacks on netlist level are hardened tremendously, since state machine signals that reach LUTs are very common, i.e. our integration is stealthy. Furthermore, by using multiple opaque predicates an attacker might not be able to fully remove the watermark. This can be especially helpful if the watermark is embedded with redundancy.

6 Conclusion

In this work we presented a novel technique to instantiate opaque predicates in hardware on HDL level. We described how existing FSMs or counters can be used to implement obfuscated constants without adding any additional gates. For situations where no existing circuitry can be used, we presented and compared three algorithms to automatically generate opaque predicates disguised in new FSM-like structures. We substantiated the stealthiness of our approach by taking the viewpoint of an attacker and performed two case studies on an FPGA to analyze the impact of our techniques on the overall design size and latency. The results showed that it is possible to implement our opaque predicates without additional FFs and only small overhead in LUTs of approximately 1%-2.2% while not impacting latency. Finally we showed how our technique is able to mitigate the open problem of netlist-level removal attacks for the LUT-based IP protection by Schmid et al. [SZT08]. To the best of our knowledge, by instantiating opaque predicates using existing FSMs or counters, our approach is more stealthy than previous work and introduces less overhead.

Acknowledgments

References

[AK07]   Yousra Alkabani and Farinaz Koushanfar. Active Hardware Metering for Intellectual Property Protection and Security. In *USENIX Security Symposium*, 2007.

[BGI+01]   Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit
              Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating
              programs. In *Annual International Cryptology Conference*, pages 1–18.
              Springer, 2001.

[BHBN14]   Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam
              Narasimhan. Hardware Trojan Attacks: Threat Analysis and Countermea-
              sures. *Proceedings of the IEEE*, 102(8):1229–1247, 2014.

[CB09]   Rajat Subhra Chakraborty and Swarup Bhunia. HARPOON: An
              Obfuscation-Based SoC Design Methodology for Hardware Protection.
              IEEE Trans. CAD of Integrated Circuits and Systems, 28(10):1493–1502,
              2009.

[Cla14]   Peter Clark. Cadence breaks into top four in semi IP core ranking. *EE
              Times Europe*, April 2014. Accessed on 12.12.2017.

[CPG+06]   Encarnación Castillo, Luis Parrilla, Antonio Garcia, Antonio Loris, and
              Uwe Meyer-Bäse. Ipp watermarking technique for ip core protection on
              fpl devices. In *Field Programmable Logic and Applications, 2006. FPL'06.
              International Conference on*, pages 1–6. IEEE, 2006.

[CT99]   Christian Collberg and Clark Thomborson. Software watermarking: Models
              and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-
              SIGACT symposium on Principles of programming languages*, pages 311–
              324. ACM, 1999.

[CT02]   Christian S. Collberg and Clark Thomborson. Watermarking, tamper-
              proofing, and obfuscation-tools for software protection. *IEEE Transactions
              on software engineering*, 28(8):735–746, 2002.

[CTL98]   Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing
              cheap, resilient, and stealthy opaque constructs. In *Proceedings of the
              25th ACM SIGPLAN-SIGACT symposium on Principles of programming
              languages*, pages 184–196. ACM, 1998.

[CTL03]   Christian Sven Collberg, Clark David Thomborson, and Douglas Wai Kok
              Low. Obfuscation techniques for enhancing software security, December 23
              2003. US Patent 6,668,325.

[CWP+09]   Rajat Subhra Chakraborty, Francis Wolff, Somnath Paul, Christos Pa-
              pachristou, and Swarup Bhunia. MERO: A statistical approach for hardware
              Trojan detection. In *Cryptographic Hardware and Embedded Systems-CHES
              2009*, pages 396–410. Springer, 2009.

[DPMDBG06]   Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Gi-
              acobazzi. Opaque predicates detection by abstract interpretation. In
              *International Conference on Algebraic Methodology and Software Technol-
              ogy*, pages 81–95. Springer, 2006.

[FT03]   Yu-Cheng Fan and Hen-Wai Tsao. Watermarking for intellectual property
              protection. *Electronics Letters*, 39(18):1316–1318, 2003.

[FWS+18]   Marc Fyrbiak, Sebastian Wallat, Pawel Swierczynski, Max Hoffmann, Se-
              bastian Hoppach, Matthias Wilhelm, Tobias Weidlich, Russell Tessier, and
              Christof Paar. Hal-the missing piece of the puzzle for hardware reverse engi-
              neering, trojan detection and insertion. *IEEE Transactions on Dependable
              and Secure Computing*, 2018.

[G+14]   A. Gascón et al. Template-based circuit understanding. In *FMCAD*, pages 83–90, 2014.

[GGKL08]   Marcin   Gogolewski,   Marcin   Gomuᵃkiewicz,   Jarosᵃaw Grza±lewicz Przemysᵃaw Kubiak, and Mirosᵃaw Kutyᵃowski Anna Lauks. Kleptographic attaks on e-auction schemes. *Tatra Mt. Math. Publ*, 41(47):47–64, 2008.

[GKK+06]   Marcin Gogolewski, Marek Klonowski, Przemysław Kubiak, Mirosław Kutyłowski, Anna Lauks, and Filip Zagórski. Kleptographic attacks on e-voting schemes. In *Emerging Trends in Information and Communication Security*, pages 494–508. Springer, 2006.

[isc]   The ISCAS-89 Benchmark Circuits. Online. http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/verilog/.

[KLT17]   Robin Kwant, Tanja Lange, and Kimberley Thissen. Lattice klepto: Turning post-quantum crypto against itself. Cryptology ePrint Archive, Report 2017/1140, 2017. https://eprint.iacr.org/2017/1140.

[L+12]   W. Li et al. Reverse Engineering Circuits Using Behavioral Pattern Mining. In *IEEE HOST*, pages 83–88, 2012.

[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.

[LJL13]   NY Louis Lee and PN Johnson-Laird. A theory of reverse engineering and its application to boolean systems. *Journal of Cognitive Psychology*, 25(4):365–389, 2013.

[MBCB+11]   Uwe Meyer-Bäse, Encarni Castillo, Guillermo Botella, L Parrilla, and Antonio Garcia. Intellectual property protection (IPP) using obfuscation in C, VHDL, and verilog coding. In *Independent Component Analyses, Wavelets, Neural Networks, Biosystems, and Nanoengineering IX*, volume 8058, page 80581F. International Society for Optics and Photonics, 2011.

[MXWW15]   Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 757–768. ACM, 2015.

[PHK17]   Khoa Dang Pham, Edson Horta, and Dirk Koch. BITMAN: A Tool and API for FPGA Bitstream Manipulations. In *DATE*, pages 894–897, 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.

[Qui52]   Willard V Quine. The Problem of Simplifying Truth Functions. *The American Mathematical Monthly*, 59(8):521–531, 1952.

[RKK14]   Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A Primer on Hardware Security: Models, Methods, and Metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.

[rm:17]   Semiconductor IP Market by Design IP (processor IP, interface IP, memory IP), Source (royalty and licensing), vertical (consumer electronics, telecom, industrial, automotive, commercial), and Geography - Global Forecast to 2023. *Research And Markets*, October 2017.

[S+10]   Y. Shi et al. A Highly Efficient Method for Extracting FSMs from Flattened Gate-Level Netlist. In *ISCAS*, pages 2610–2613, 2010.

[S+12]   Y. Shi et al. Extracting Functional Modules from Flattened Gate-Level Netlist. In *ISCIT*, pages 538–543, 2012.

[S+13]   P. Subramanyan et al. Reverse Engineering Digital Circuits Using Functional Analysis. In *DATE*, pages 1277–1280, 2013.

[S+14]   P. Subramanyan et al. Reverse Engineering Digital Circuits Using Structural and Functional Analyses. *IEEE Trans. Emerging Topics Comput.*, 2(1):63–80, 2014.

[SFK+16]   Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. Interdiction in Practice—Hardware Trojan against a High-Security USB Flash Drive. *Journal of Cryptographic Engineering*, pages 1–13, 2016.

[SI14]   V. Sergeichik and A. Ivaniuk. Implementation of Opaque Predicates for Fpga Designs Hardware Obfuscation. *Journal of Information, Control and Management Systems*, 12, 01 2014.

[STBF17]   Bicky Shakya, Mark M Tehranipoor, Swarup Bhunia, and Domenic Forte. *Introduction to Hardware Obfuscation: Motivation, Methods and Evaluation*, pages 3–32. Springer, 2017.

[SZT08]   Moritz Schmid, Daniel Ziener, and Jurgen Teich. Netlist-level IP protection by watermarking for LUT-based FPGAs. In *ICECE Technology, 2008. FPT 2008. International Conference on*, pages 209–216, 2008.

[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.

[WFSP17]   Sebastian Wallat, Marc Fyrbiak, Moritz Schlögel, and Christof Paar. A Look at the Dark Side of Hardware Reverse Engineering – A Case Study. In *IVSW*, 2017.

[WSS13]   Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. FANCI: Identification of Stealthy Malicious Logic using Boolean Functional Analysis. In *ACM CCS*, pages 697–708, 2013.

[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.

[YY07]   Adam Young and Moti Yung. Space-efficient kleptography without random oracles. In *Information Hiding*, pages 112–129. Springer, 2007.

[ZYW+13]   Jie Zhang, Feng Yuan, Linxiao Wei, Yannan Liu, and Qiang Xu. VeriTrust: Verification for Hardware Trust. In *ACM/EDAC/IEEE (DAC)*, pages 1–8, 2013.