SWAN: Mitigating Hardware Trojans with Design Ambiguity

Timothy Linscott Pete Ehrett Valeria Bertacco Todd Austin
University of Michigan, Ann Arbor
`{timlinsc, wpehrett, valeria, austin}@umich.edu`

ABSTRACT

For the past decade, security experts have warned that malicious engineers could modify hardware designs to include hardware backdoors (trojans), which, in turn, could grant attackers full control over a system. Proposed defenses to detect these attacks have been outpaced by the development of increasingly small, but equally dangerous, trojans. To thwart trojan-based attacks, we propose a novel architecture that maps the security-critical portions of a processor design to a one-time programmable, LUT-free fabric. The programmable fabric is automatically generated by analyzing the HDL of targeted modules. We present our tools to generate the fabric and map functionally equivalent designs onto the fabric. By having a trusted party randomly select a mapping and configure each chip, we prevent an attacker from knowing the physical location of targeted signals at manufacturing time. In addition, we provide decoy options (canaries) for the mapping of security-critical signals, such that hardware trojans hitting a decoy are thwarted and exposed. Using this defense approach, any trojan capable of analyzing the entire configurable fabric must employ complex logic functions with a large silicon footprint, thus exposing it to detection by inspection. We evaluated our solution on a RISC-V BOOM processor and demonstrated that, by providing the ability to map each critical signal to 6 distinct locations on the chip, we can reduce the chance of attack success by an undetectable trojan by 99%, incurring only a 27% area overhead.

CCS CONCEPTS

• **Security and privacy → Tamper-proof and tamper-resistant designs**; *Hardware security implementation*;

ACM Reference Format:
Timothy Linscott, Pete Ehrett, Valeria Bertacco, and Todd Austin. 2018. SWAN: Mitigating Hardware Trojans with Design Ambiguity . In *IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD '18), November 5–8, 2018, San Diego, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3240765.3240854

1 INTRODUCTION

As the number of leading-edge silicon manufacturers shrinks, more and more hardware designers are turning to third-party manufacturers to fabricate their chips. However, recent work has warned that, if unvetted third-party silicon fabs employ malicious engineers, they could modify a chip's layout to introduce a hardware backdoor, a.k.a., a **trojan**. Trojans are typically small silicon modules, even as small as one gate or less [1, 2], that monitor a system's operation at runtime, lying dormant until a specific sequence of events enables them to pass control of the system's operation (*e.g.*, by providing superuser rights) to an attacker.

Despite many advances in protections against hardware trojans, a systematic and general defense approach remains elusive. Obfuscation defenses [3, 4] attempt to hide the role of specific signals from the attacker. However, recent advances in obfuscation have been matched by advances in reverse-engineering technologies, and an attacker who can successfully reverse-engineer an obfuscated design is likely to succeed at planting a trojan without detection. Logic analysis [5] and physical inspection [6, 7] provide a variety of tools to identify trojans' trigger conditions and detect whether the fabricated chip differs from the expected layout. However, some recent trojan designs [1, 2] can elude either approach. On-chip and off-chip checkers are a promising alternative to inspection techniques, as they can detect trojan activity at runtime, when the attack is active, but this approach is only effective if the attacker cannot tamper with the checkers or their inputs and outputs.

In this work, we tackle these problems head-on by assuming an exceptionally powerful attacker. Our attacker has compromised the manufacturing house and has full knowledge of the chip and its defenses. As such, the attacker can make the final modifications to the design before manufacturing, and is completely free to add, modify, or remove logic. To counter such an adversary, we propose a novel defense against trojans, called SWAN (Security With Ambiguous Netlists). SWAN ensures that designers (instead of attackers) are always in charge of the final move in the design process, by including a small amount of configurable logic to be finalized by a trusted party after manufacturing. For this purpose, we use a ***one-time programmable fabric as a means to hide security-critical signals from the attacker, protecting both the critical logic and any checkers that are monitoring it.*** Figure 1 overviews our approach: while the original chip design is trusted, an attacker may insert a trojan during the manufacturing process. When the chip is assembled by a trusted party (or by the design house), the one-time-programmable fabric portion receives a randomly selected

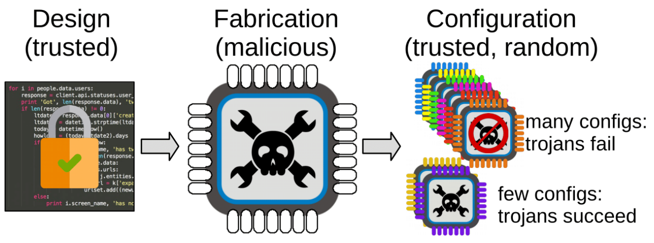


Figure 1: SWAN protects security-critical modules by replacing them with one-time programmable logic. As a result, an attacker cannot know at manufacturing time the physical location of specific security resources, minimizing their chance of success.

configuration that implements the security-sensitive logic and its checkers. Note that designers will have developed many distinct and functionally equivalent configurations that can all be accommodated by the programmable fabric. Complex trojans can identify the configuration, but their large silicon footprint makes them prone to detection. In contrast, simple trojans can go undetected, but designers can make their chances of success vanishingly small. Moreover, for each configuration, we recycle unused logic as 'canaries', which expose an attack attempt when the canary's output deviates from its expected value. Our semi-automated toolchain provides a streamlined flow for developers to apply these protections to their designs at the logic-unit level, and rapidly build programmable fabrics out of arbitrary SystemVerilog descriptions. In summary, our contributions are as follows:

• We present SWAN, an automatically generated, one-time programmable architecture that serves to hide from the attacker the physical location of crucial design elements during manufacturing, even if the design is completely reverse-engineered.

• We developed tools to generate i) one-time programmable fabrics optimized for the logic that we need to protect, and ii) the many equivalent configurations that must be mapped to them. We use these tools to evaluate SWAN on a RISC-V out-of-order core, protecting five security-critical components.

• We examine the area and power overheads of SWAN and find them to be moderate even for highly secure applications, and significantly lower than an FPGA-based solution. For instance, by mapping each critical signal to 6 distinct locations, we incur only 27% silicon area and 5% power overhead. This level of protection reduces the chance of a successful attack by an undetectable trojan by 99%.

2 THREAT MODEL

Before we present the structure of our solution, we must specify the threat model that motivates our defense mechanism and design choices. In our threat model, the attacker's goal is to disrupt the chip's security properties. This goal could include toggling a privilege bit during user-mode execution, removing access restrictions to privileged pages in memory, or overriding the program counter to manipulate control flow. In this work, we do not directly attempt to protect against denial-of-service attacks or side-channel leakage attacks, as these require different classes of defenses. In contrast, the defender's goal is to ensure that, with high probability, any attempted attack will result in either the attack being detected and thwarted, or the system halting with no violation of its security guarantees. Fortunately, physical inspection techniques are improving, and today, they are able to detect any sufficiently large (~$90\mu m^2$) modification to the layout [6, 7]. Thus, the defender's secondary goal is to ensure that any trojan capable of circumventing their defense system will have such a large silicon footprint that it would be detected by physical inspection.

Today, the design and manufacturing of a chip often occur at different companies, limiting the amount of oversight over the manufacturing process by the designing company. Thus, it is reasonable to assume that a malicious manufacturing engineer, an attacker, could make arbitrary changes to the design's layout without the design team's knowledge. Given recent advances in reverse-engineering technology, and the possibility that design files could make their way out of the design house, we must also assume that the attacker may have access to all of the design team's register-transfer level (RTL) descriptions and EDA tools. With sufficient time and resources, an attacker could reach a perfect understanding of the function of every transistor in the design, be fully aware of every protection included in the design, and have acquired every possible configuration that was developed for the one-time programmable fabric. The same attacker would also have knowledge of the post-silicon tests planned for the chip; thus, they can select trojan triggers that will not be exposed during testing (*e.g.*, in [1], the trojan could only be activated at high temperatures). In other words, we could assume that the attacker is omniscient. But even with these extreme assumptions, the attacker cannot know which configuration will be selected for the fabric after manufacturing.

Fortunately, given our premises, we can trust that the products of the design house are uncompromised: design files, post-silicon tests, and software layers. SWAN does indeed require access to a golden, untampered copy of the chip's RTL description, from which it can generate a trustworthy one-time programmable fabric and all of its related configurations. We assume that all software layers, including the BIOS, OS, and other privileged code, are uncompromised: if the attacker could modify those, they would not need to invest in devising a hardware trojan to gain control of the system.

To summarize, our defense targets an attacker who: i) is attempting to compromise hardware security guarantees, such as privilege rings; ii) is knowledgeable of all design details, but cannot predict which configuration will be mapped on the one-time programmable fabric at assembly time; and iii) can make arbitrary changes to the design after it leaves the design house. The attacker is defeated when they cannot circumvent SWAN's defenses without being detected by on-chip checkers or by physical inspection.

3 SECURITY GOALS

In our threat model, we assume that the attacker is able to avoid detection by post-silicon functional tests. Thus, there must be alternate ways, after deployment, for a trojan to be detected when it becomes active (on-chip runtime checkers, for instance). In this context, an effective defense against hardware trojans must satisfy a number of key properties in order to be robust against our powerful attacker. Such a defense mechanism must:

(1) be *tamper-proof*. The attacker must not be able to disable the defense without detectable side-effects.
(2) be able to *accurately obverse system state*. The attacker must not be able to hide their actions from the defense.
(3) *not be suppressible*. When the defense identifies an attack, its mechanisms to restore the system to a safe state must be tamper-proof as well.
(4) be *verifiable*. If an attacker attempts to compromise the defense, this attempt should be recognizable at the software layer. The defense cannot be invisible to the software.
(5) be *low-cost*, so as to be practical and viable.

If the defense mechanism were implemented in some kind of field-programmable logic, the final design would remain undefined at manufacture-time, its implementation would only be selected at assembly time, and it could vary between chips. In such a scenario, because the fabric has yet to be configured at manufacture-time, the attacker would not know which physical resource will implement the attack-target design element(s), making the design of a trojan

Table 1: The RISC-V BOOM modules selected for SWAN protection.

Module	Description	Consequence if compromised	Area (μm^2)	% of chip area
CSRFile	Control status register file	Privilege escalation	24,546	2.67%
DecodeUnit	Instruction decoder	Execute privileged instructions, inject code	1,692	0.18%
Frontend	Program counter manager	Control flow attacks	3,623	0.39%
PTW	Page table walker	Access privileged pages	11,164	1.21%
TLB	Virtual to physical address translation	Access privileged pages, expose physical addresses	11,400	1.24%
Total			52,427	5.69%

challenging, to say the least. Even if the attacker had perfect knowledge of all possible configurations, they still would not know which one was going to be selected for a given chip. Finally, a key goal of SWAN is to provide a design and set of configurations such that, if the attacker added a trojan sufficiently complex to be able to evaluate precisely which configuration-time mapping was selected, then the trojan would likely be so large that it would be easily noticed by physical inspection techniques. Thus, logic implemented on this programmable fabric can be made probabilistically *tamper-proof*.

Note that any online checker deployed to detect active trojan activity must also be mapped to the same programmable logic; otherwise, the checker could be identified by the trojan and silenced. Moreover, such checkers are also able to *accurately observe the system state*, since they can monitor the state without leaving the protected programmable fabric. Our third point specifies that, when the defense mechanism detects an attempted attack, it must be able to restore the system to a safe state. Again, this goal can be only achieved if the attack-response system is also encapsulated within the programmable fabric, so that an attacker cannot silence it.

Finally, field-programmability provides opportunities to expose hardware defenses to the software layer, making them *verifiable* by creating side channels that would be disrupted by an attacker modifying the fabric. If the properties of the logic can be made to vary between configurations in a software-detectable way, then the software can verify that the fabric has been configured correctly, demonstrating that the attacker has not hijacked it with their own configuration. The next section discusses how we provide the capabilities above, while maintaining a *low-cost* profile for SWAN, by building a custom, LUT-free, one-time programmable architecture.

4 SWAN ARCHITECTURE

There are many types of field-programmable logic, and not all are well-suited to our goals. First, reconfigurability is expensive in terms of area and power consumption, motivating our choice of a one-time programmable fabric. Second, the fabric must support enough distinct mappings of each component to prevent the attacker from easily guessing a probable configuration. SWAN's programmable fabric is automatically generated based on the target security module selected. To minimize overheads, only security-critical modules are mapped to this fabric.

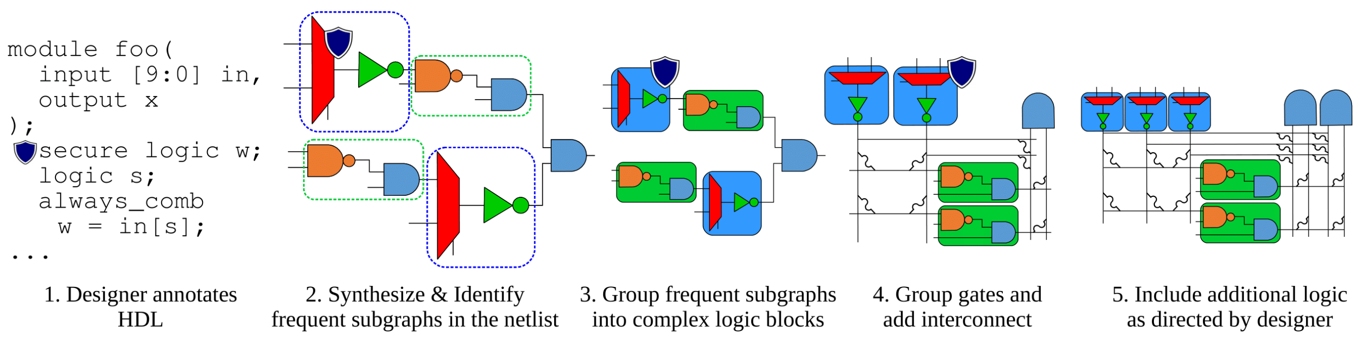
Because prior work has targeted specific parts of CPU cores with hardware trojans [1], we examine the RISC-V BOOM out-of-order core [8] as a case study to identify modules critical to maintaining security. First, we want to ensure that the privilege rings cannot be compromised, so we secure every module that accesses the privilege bit: the control status register file, the memory management unit, the instruction decoder, and the page table walk logic. Second, we must protect the execution's control flow, so we also secure the program counter and associated front-end logic. Table 1 describes these modules, the attacks that could be perpetrated by a trojan gaining control over them, and their contributions to chip area. We leave application of our proposed techniques to additional modules, or to custom accelerator hardware, for future work.

4.1 Secure Programmable Logic

SWAN's programmable fabric is optimized to increase the number of possible mappings of the target modules to the fabric, in order to thwart an attacker who wishes to identify the signals that should be connected to a trojan that they plan to deploy. We attain this goal at the expense of flexibility, since our fabric can only implement one target design. The fabric comprises sets of identical fixed-function logic blocks, instead of LUTs, connected by one-time programmable crossbars. Each logic block can be configured by a trusted party after manufacturing to drive a different part of the circuit, effectively mapping one logic block from the design to any of several possible physical blocks. This mapping is configured by setting the fuses embedded in the crossbars. Our fabric generator creates the one-time programmable fabric using only gates that are present in the RTL description of the design, and then adds canary logic (Section 4.3) and the crossbars that serve as the programmable interconnect (which dominate the fabric's overheads).

Our automated toolchain handles the generation of the fabric, starting from an initial description using a hardware description language (HDL) – SystemVerilog in our evaluation – of the modules to be protected. Figure 2 highlights the main steps of this toolchain flow. We allow the programmer to include specific pre-processor directives in the HDL description to guide the SWAN toolchain to use specific security extensions. Specifically, we expect the programmer to mark security-critical signals as `secure`, and to include a secure rollback system that can handle and recover from a situation where a canary detects a potential trojan attack, using a special input marked `canary`. SWAN's pre-processor scans the HDL for these directives and logs them for use in the main toolchain. The target module is then synthesized, and the generated netlist is converted to a graph. GraMi, an open-source graph analysis tool [9], mines this graph for frequent subgraphs to identify good candidates for the building blocks of our fabric (step 2 in Figure 2). While a standard coarse-grained reconfigurable architecture would use large adder and multiplier units as building blocks, our fabrics use only small clusters of gates, because SWAN primarily targets control logic. To select which of the mined subgraphs will become the building blocks of our final fabric, we greedily choose those with the most intra-block wiring. This limits the size of the inter-block crossbars required to support the fabric's programmability, thus containing area costs. Once selected, each subgraph occurrence in the netlist is replaced with a primitive logic block (an indivisible cluster of gates) as shown in step 3 of Figure 2.

Next, the newly obtained netlist is analyzed to identify functionally equivalent gates and logic blocks that are in close proximity to each other in the graph. These blocks are grouped into sets (step 4 in Figure 2), to provide opportunities for generating multiple equivalent mappings. In the final manufactured fabric, all blocks in the same set will be completely indistinguishable from the attacker's perspective. The toolchain further guarantees that each logic block



```
module foo(
  input [9:0] in,
  output x
);
secure logic w;
  logic s;
  always_comb
   w = in[s];
...
```

1. Designer annotates HDL
2. Synthesize & Identify frequent subgraphs in the netlist
3. Group frequent subgraphs into complex logic blocks
4. Group gates and add interconnect
5. Include additional logic as directed by designer

Figure 2: The **SWAN toolchain** considers (1) user-annotated HDL and generates a custom one-time programmable fabric, by (2) synthesizing it, (3) grouping identical logic blocks, and (4) providing a flexbile interconnect to permute their roles. (5) Finally, it enhances security with camouflaging and canaries.

from the original netlist has a user-determined minimum number of physical resources onto which it can be mapped.

At this point, every logic block in the design can be mapped to a fixed number of locations in the physical fabric. The design's netlist and structure are used to generate the interconnect: each group of blocks should be capable of selecting its inputs from all and only from the groups of blocks that could drive it. At this stage, we add programmable crossbars into the netlist to enable each block in a set to serve the role of any block in its own set. Next, the tool selects which physical resources will be used to implement the design and determines the exact configuration needed to map all the components onto the fabric. The finalized fabric is provided as a netlist (also in SystemVerilog in our evaluation), and can be used in place of the original module without any modification. Verification of the generated fabric and each of its configurations can be accomplished with an equivalence checker to show that the configured netlist is equivalent to the original design. Since each configuration is topologically identical to the original netlist, the equivalence checker's task reduces to simple graph-matching.

4.2 Signal Camouflaging

Programmable logic offers a number of unique opportunities to enhance security even further: as described below, we exploited the nature of our defense to build features in our toolchain that are both optional and fully under the control of the designer. Specifically, we provide a mechanism for the designer to enable certain signals to be mapped to many locations, making attacks on those signals more challenging for an attacker. We provide this camouflage functionality by allowing the programmer to tag signals with a **`secure`** pre-processor directive. This tag ensures that gates that output or use a critical signal can be mapped to a large number of possible physical locations in the final fabric. When the annotated HDL is compiled, these gates are identified in the netlist and tagged. Finally, during the grouping phase of the fabric generation flow (step 4 in Figure 2), sets containing these secure gates are made larger to ensure a higher replication of the logic blocks they include.

As an example of where camouflaging can be useful, consider the control status register file of a processor. While all the logic in this register file is important, the privilege register is the most desirable target for an attacker. If a gate in the design can be mapped to one of only a small number of locations in the physical fabric, an attacker who knows this fact can build a stealthy trojan that picks out its target from among those few gates. However, by tagging the

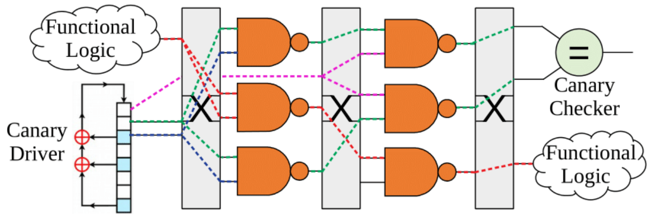


Figure 3: Canaries are identical chains of logic selected from among the unused logic in the programmable fabric. They share a common driver, so their outputs match if untampered, as verified by auto-generated checkers.

output of this register for camouflaging, our toolchain will grow the interconnect so that a larger number of identical physical gates will all able to be configured into acting as the privilege register. Note that camouflaging should be used parsimoniously, as increased interconnect size means higher overheads. Thus, camouflaging is an effective knob to trade off between cost and security.

4.3 Canary Logic

While signal camouflaging prevents attackers from succeeding every time, a truly secure system also needs a way to detect if trojans are making repeated, brute-force attempts at attacks, so as to stop them before they guess correctly. Thus, we provide a mechanism to repurpose unused logic as canaries. Like their counterparts in software security, called stack canaries [10], logic canaries are designed solely to detect attacks that fall onto them. This prevents attackers from randomly placing a malicious circuit and triggering it on every chip until they find a configuration where their target is mapped to the trojan's location. Canaries guard against this brute-force approach by detecting when an attempted attack fails—as it is likely to do with most configurations. To incorporate canaries into a design, the toolchain includes additional resources (logic blocks or gates) in the fabric, allowing more physical mapping opportunities for logic blocks from the source design. These extra gates are added during the final step (step 5 of Figure 2) to sets of gates marked **`secure`** (Section 4.2) or those that do not meet the user's desired minimum replication rate.

Figure 3 shows how canaries are deployed in SWAN. The canaries are arranged into identical chains of unused gates, and the chains are driven by an automatically sized linear feedback shift register (LFSR) — a.k.a. a *canary driver* — built into the SWAN fabric. Thus, every canary will have its functionality rigorously tested for all possible inputs, and this testing can continue (and should, to detect latent triggers) as long as the chip is active. The outputs of similar canary chains are compared via automatically generated checking logic within the fabric. If an attacker modifies one canary block, its output will no longer match that of the other canaries, and the checking logic will raise an alarm. The programmer can access this alarm signal by including an input flagged with `canary` to the SWAN portion of the design, and can then build a recovery system to respond to the detected attack. The intermediate signals in the canary checker logic further allow the programmer to gain insight into which regions of the design are potentially under attack, thereby helping to pinpoint the trojan's location.

If an attacker attempts a brute-force attack on the fabric by flipping random signals in hope of guessing the current mapping of their target, they may cause unintended errors in the process by flipping signals unrelated to their attack. However, without canaries, these errors may not be recognized as an attack in progress on their own. In contrast, by including canaries, we can accurately report such errors for what they are: a trojan attack in progress.

4.4 Programmer-Defined Side-Channels

We provide two features that enable programmers to introduce low-overhead flexibility in the fabric, allowing them to create side-channels they can test. In order for the programmable logic to prove it was programmed only by the trusted assembly house, it must exhibit different behaviors on different configurations. For example, the decoder could interpret a secret opcode as an add instruction in one configuration and as a sub in another. Thus, an attacker who disables the programmable nature of the fabric is detected when the side-channel does not demonstrate its expected properties.

The first way we introduce this low-overhead flexibility is with configuration-defined constants. These allow programmers to directly drive gates in the design from fixed logic values, resulting in different behavior between configurations. On a larger scale, we allow the programmer to vary the parameters of a submodule that they want to make flexible. Then, a design is synthesized for every possible assignment of the parameter, producing many possible netlists. These netlists are merged into a single fabric that supports any of the designs generated, while optimizing for cost.

5 EVALUATION

We evaluated our design along two metrics: first, SWAN's impact on the system's area, power, and clock frequency; second, security. Because different applications require different security guarantees, we provide a designer with the flexibility to trade security for cost.

5.1 Framework

We implemented our design using a 1-wide RISC-V BOOM (out-of-order) core as a baseline. At the time of this writing, BOOM did not have an internal cache hierarchy. We used CACTI [11] to estimate the overhead of including 32KB L1 instruction and data caches and an 8-way, 256KB L2 cache, and factored their power and area cost into the overall results. The control status register file, decoder, front-end and PC logic, page table walker, and TLB modules (see Table 1) were replaced with our custom-generated one-time programmable fabric. In the baseline RISC-V BOOM core, these modules represent about 6% of the core area, excluding caches. We used Synopsys Design Compiler with the NanGate 45nm cell library to synthesize our designs and determine their area, static/dynamic

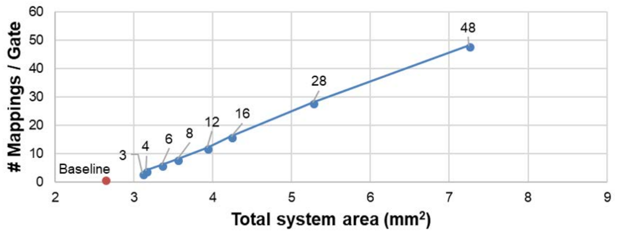


Figure 4: The # of possible mappings of a gate to the fabric can be increased by adding more canaries and interconnect, raising area cost.

power, and critical path delays, as well as the footprint of trojans embedded in SWAN. For comparison, we also implemented the target components on a Xilinx UltraScale+ ZU9EG FPGA.

In testing our one-time programmable fabric, we explored the effects of two key parameters. First, we varied the number of locations to which each gate can be mapped, along with the amount of signal camouflage used, so that gates could be mapped to anywhere from 2 to 48 different locations on the fabric. Second, we varied the amount of extra logic on the chip that could be used as canaries to boost the chance of detecting an attack attempt.

As the baseline design, we use the BOOM core without SWAN, synthesized as per Section 5.1. We substituted out the components we wanted to protect with SWAN, testing different amounts of signal camouflage, and recomputing the total area, power, and critical path of the full system each time. Note that, because SWAN changes only the implementation of the core (not its underlying architecture), the sole determinant of SWAN's application-level performance impact is its critical path delay.

Figure 4 plots how the core's area increases with the number of distinct locations to which each secured component can be mapped. We do not plot critical path delay or power, as these remain roughly constant across different SWAN solution points (see Table 2).

5.2 Overheads

As a point of comparison, FPGAs could provide a natural source of the kind of security through reconfigurability we are interested in. But since they are designed for fine-grained flexibility instead of security, they cannot make comparable guarantees to those SWAN provides with signal camouflage and canary logic. Table 2 shows how using on-chip FPGA logic (with no camouflaging) to replace the target components would impact the area, power, and delay of the RISC-V baseline. The table also reports results for two SWAN implementations with 6 and 12 mappings per gate, which provide a 99% and 99.9% chance of detecting a hardware trojan, respectively. Because of our fine granularity in defining logic primitives, SWAN experiences a notable increase in path delay (up to 82%) over the baseline, due to signals traversing a significant number of fused crossbars along each path. However, power overhead is low and roughly constant across all SWAN solutions we evaluated, because we power-gate the programming logic and the fraction of exercised logic remains constant across configurations. Moreover, the comparison to the FPGA implementation is highly favorable: the 6-mapping SWAN solution has a 32% shorter critical path while consuming 31% less power and 41% less area. Note that the 12-mapping solution has little to no more power or timing overhead than the 6-mapping solution; only area is affected.

Table 2: FPGA and SWAN trojan defense overheads.

	Baseline	FPGA (16nm)	SWAN (99% detection)	SWAN (99.9% detection)
Power (mW)	578	889	602	604
Area (mm^2)	2.63	5.70	3.35	3.93
Crit path (ns)	4.5	12.1	8.2	8.2

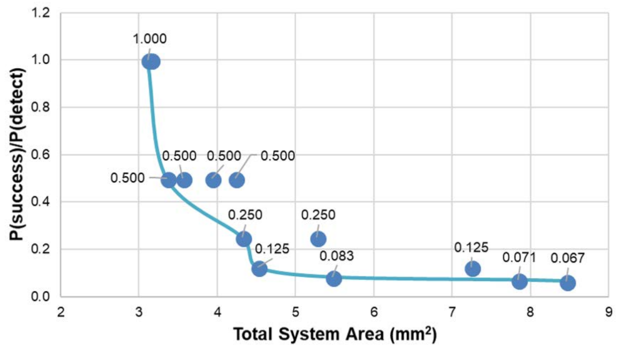


Figure 5: Ratio of trojan success over trojan detection vs. SWAN fabric area. Increasing the amount of canaries in SWAN causes the attacker's odds of succeeding without detection to fall rapidly. Thus, a blind guess approach is likely to be quickly thwarted.

5.3 Security Analysis

To properly analyze the security of the system and determine an optimal set of parameters for generating the SWAN fabric, we assume that designers have identified all those wires within the critical module which, if toggled by an attacker, would fail a security assertion, and that they have flagged those signals with the `secure` directive (Section 4.2). We also assume that the attacker's goal is to flip a number of those signals. The attacker has two possible strategies: (i) attempt to guess the current configuration, or (ii) analyze the configuration of the crossbars to determine the mapping of their target gate(s). We consider here an example attack scenario where the attacker wants to flip the privilege register bit, to escalate the permissions of their attack software code. Since SWAN protects both security-critical logic and its checkers together, we assume that the attacker will also need to disable a checker to hide their attack. Thus, the attacker must flip a total of 2 bits.

Blind Guess Attack Approach. The simplest attack against SWAN is for an attacker to blindly guess the location of their target by placing a single copy of their trojan in one location and then activating it to see whether they guessed correctly. While such a trojan would be extremely light-weight, since it does not attempt to circumvent SWAN, it risks exposure by accidentally triggering canary logic. Thus, for such an attack to be worthwhile, the risk of detection must be modest in comparison to the chance of success. SWAN enables designers to control these two probabilities by adjusting the amount of signal camouflage and the number of canaries used in the fabric. In Figure 5, we compare a variety of possible SWAN designs, and provide a Pareto-optimal set of parameters that minimize the ratio of trojan success vs. detection, for a range of silicon areas of the SWAN module. For instance, the solution labeled 1.000 provides 3 mappings for each gate, one of which is a canary. Thus, a trojan has 1/3 chance of succeeding and 1/3 chance of being detected. Even with a very modest area increase, the ratio becomes

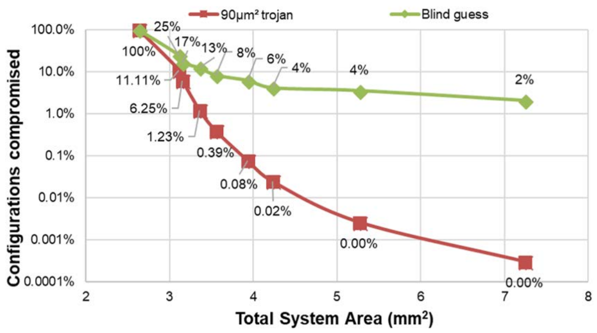


Figure 6: % of chips compromised by different trojan approaches as the SWAN fabric area increases. Blind-guess trojans may be detected by canaries when they are unsuccessful, while analytical trojans must limit their analysis to maintain a low area profile: at 6 locations per secure gate, their chance of success is below 1%.

very small, presenting a prohibitive obstacle for attackers, since detection means a high likelihood of exposure for the malicious entity inside an untrusted fab.

Analytical Attack Approach. Given the likelihood of detection with a blind guess attack, an attacker could pursue an approach that tests how fuses have been set before activating the trojan, to make sure that the trojan does not trigger a canary. This approach would allow the trojan to go undetected at the cost of only compromising the small fraction of manufactured chips that were configured exactly as the attacker guessed when placing the trojan. Even better, to attack all configurations, they could place trojans at each potential location of their target bit and use test logic to evaluate which configuration has been set before activating only the correct trojan. The downside of this approach is that it requires deploying a significant amount of logic to monitor many fuses in multiple locations. This additional silicon makes the trojan prone to detection by physical inspection of the manufactured chip. Multiple related works report how relatively small regions of spurious logic could be identified in a silicon chip: in particular, [7] demonstrated an imaging technique that classifies individual gates in a 45nm chip, at an accuracy of 85%, and [6] used a watermarking technique to identify trojans smaller than 90μm^2, also on a 45nm chip. These studies place an upper bound on the number of locations that an attacker can simultaneously monitor.

Instead of placing a trojan on every possible location, the attacker could stop placing additional trojans when the total area footprint of the trojan exceeds 90μm^2. By staying below this threshold, they can avoid physical detection. As before, once each trojan has analyzed the relevant fuses, it determines whether to toggle the output of the gate to which it is connected or to remain dormant. However, as the number of locations in the fabric where a component can be placed increases, the fraction of chips where this approach can succeed decreases.

To evaluate the practicality of this approach, we first identified all the locations where a block protected by signal camouflage could be mapped. For as many of these as possible within 90μm^2, the attacker would then deploy one bit-flipping trojan and a number of monitors, in such a way as to check the fewest fuses, and still determine whether the location under investigation hosts the bit to be attacked.

Figure 6 explores the chance of success of the two attack approaches described. In one, the attacker uses the analytical attack approach to flip a single bit in the fabric and a second bit to silence a checker elsewhere in the fabric. In the other, the attacker applies a blind guess approach to a target-signal location and a checker location. The blind-guess approach is more likely to succeed, even in highly protected SWAN setups, compared to the area-bound analytical approach. However, a failure of the blind-guess approach may correspond to an attack detection by canaries, tipping off the design house to the presence of the trojan on the entire population of chips. The plot shows that, for a SWAN solution with an area of 3.93mm^2, corresponding to mapping critical components to at least 12 distinct locations, we ensure that over 99.9% of the manufactured chips will not be compromised. Likewise, at 3.35mm^2, with 6 distinct mappings, an attacker will fail 99% of the time.

6 RELATED WORK

In this section, we provide an overview of other defenses proposed in the literature and contrast them with our work. SWAN is often composable with these defenses, as we point out in many places within the discussion. Defenses against trojans draw from either post-silicon detection, runtime detection, or obfuscation.

Post-silicon detection solutions look for trojans with microscope inspection, power analysis, or logical analysis. TeSR (Temporal Self-Referencing) [12] aids in discovering trojans via power analysis, even when there is no golden, trojan-free reference. It does this by executing many traces on the chip and looking for outliers in the expected current signature. Logical analysis often employs debugging circuitry to detect potential trojan-trigger signals. In [5], the circuitry is activated by a secure key, and allows for automatic exploration of possible states, with traces logged on the fly, to allow for immediate detection. Unlike both of these, we do not assume that testing can expose the trojan's activation trigger, since such triggers can be made arbitrarily complex. In contrast, the goal of SWAN is to force the attacker to place such a large trojan that simple physical inspection will reveal its presence.

Runtime detection includes all kinds of on-chip and off-chip checking logic that monitors for violations of the chip's security. [13] proposed a method to generate such checkers directly from the specification, ensuring high coverage of potential attack surfaces. [14] investigates tampering of runtime checkers. To counter this, they introduce an element of randomness in the checker's inputs striving to prevent the attacker from successfully issuing a trigger code that would disable the checker. Our work provides a mechanism to protect checkers and guarantee observability into the components they protect. SWAN also includes a checker of its own, the canary logic, to monitor for attempted attacks.

Obfuscation takes a cryptographic approach to preventive defense. In an obfuscated circuit, a key must be provided to correctly implement the output function. Furthermore, it is supposed to be intractable to reverse engineer the obfuscated circuit if the attacker only has a black box version of the circuit. [3] implements obfuscation by replacing a small portion of logic with FPGA-like LUTs. An attacker who wants to reverse engineer or tamper with the design would need to know the correct configuration of the LUTs, which becomes quickly intractable with increasing numbers of LUTs. A complete FPGA-like reconfigurable fabric has also been proposed. In [4], the authors use an FPGA fabric to implement instruction set randomization on a processor and protect against hardware trojans designed for code injection. Because our attacker is assumed to be omniscient, they can also overcome obfuscation. Replacing logic with LUTs on a gate-by-gate basis still leaves the design exposed to attackers who know how the LUTs will be configured, especially if there is only one configuration that correctly implements the functionality of the system. However, it is still possible to block such an attacker obfuscating the design and providing multiple mappings to a reconfigurable fabric. Our approach not only hides the functions of design blocks, it also conceals their locations. Moreover, using a complete FPGA-like fabric to implement the target components would carry very high overheads, which SWAN seeks to avoid.

7 CONCLUSIONS

We presented SWAN, an automatically generated one-time programmable architecture, created to enhance security at modest cost. Our toolchain provides several additional security features to the programmer, allowing them to re-use excess resources as checking logic, and to introduce only as much flexibility as they deem necessary. We implemented SWAN on a RISC-V core and analyzed its security properties. As the size of the SWAN fabric increases, the minimum area cost of an ideal hardware trojan limits the attacker to only targeting a few locations in the fabric. By leveraging this security vs. area trade-off, we can parameterize SWAN so that an attack capable of disabling a checker would have a 99% chance of failure. At this solution point, SWAN incurs less than 5% total power and 27% area overhead— significantly better than an FPGA implementation, which would at least double the area while providing far more limited security guarantees.

Acknowledgment. This material is based on research sponsored by the Defense Microelectronics Activity (DMEA) under agreement number H94003-11-2-1101. The United States Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon.

REFERENCES

[1] Q. Dong T. Austin K. Yang, M. Hicks and D. Sylvester. A2: Analog malicious hardware. In *S&P*, 2016.
[2] C. Paar G. Becker, F. Regazzoni and W. Burleson. Stealthy dopant-level hardware trojans. In *CHES*, 2013.
[3] H. Mahmoodi K. Gaj T. Winograd, H. Salmani and H. Homayoun. Hybrid STT-CMOS designs for reverse-engineering prevention. In *DATE*, 2016.
[4] B. Liu and B. Wang. Embedded reconfigurable logic for ASIC design obfuscation against supply chain attacks. In *DATE*, 2014.
[5] S. Paul R.S. Chakraborty and S. Bhunia. On-demand transparency for improving hardware trojan detectability. In *HOST*, 2008.
[6] B. Zhou et al. Detecting hardware trojans using backside optical imaging of embedded watermarks. In *DAC*, 2015.
[7] R. Adato et al. Rapid mapping of digital integrated circuit logic gates via multi-spectral backside imaging. *arXiv:1605.09306*, 2016.
[8] K. Asanovic C. Celio, D. Patterson. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. *Tech. Rep. UCB/EECS-2015–167, EECS Department, UC Berkeley*, 2015.
[9] S. Skiadopoulos M. Elseidy, E. Abdelhamid and P. Kalnis. Grami: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, 2014.
[10] D. Maier J. Walpole P. Bakke S. Beattie A. Grier P. Wagle Q. Zhang H. Hinton C. Cowan, C. Pu. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX*, 1998.
[11] R. Balasubramonian N. Muralimanohar and N. Jouppi. Cacti 6.0: A tool to model large caches. In *MICRO*, 2007.
[12] D. Du R.S. Chakraborty S. Narasimhan, X. Wang and S. Bhunia. TeSR: A robust temporal self-referencing approach for hardware trojan detection. In *HOST*, 2011.
[13] C. Irvine M. Bilzor, T. Huffmire and T. Levin. Security checkers: Detecting processor malicious inclusions at runtime. In *HOST*, 2011.
[14] D. Hély J. Dubeuf and R. Karri. Run-time detection of hardware trojans: The processor protection unit. In *ETS*, 2013.