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, becomes very complex, exposing it to inspection techniques due to its large silicon footprint. We evaluated our solution on a RISC-V BOOM processor and demonstrated a 99% reduction in the security resources, leading to a very low chance of success.

In this work, we propose a novel solution 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 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 configuration that implements the security-sensitive logic and its checkers. Note that the designer 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 be undetectable, but the designer can make their chance of success
vanishingly small. Moreover, for each configuration, we recycle unused logic as ‘canaries’, which expose an attack attempt when the canary output deviates from its expected value. 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 when the design had been completely reverse-engineered.
- We develop 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 and 5% power overheads. This same level of protection leads to a 99% reduction in the likelihood of successful attack by an undetectable trojan.

2 THREAT MODEL

Before we present the structure of our solution, we must motivate our design choices. To this end, it is important that we specify the threat model that motivated our defense mechanism. Every defense mechanism is designed to combat an attacker who is assumed to have a certain degree of knowledge of the system’s defense mechanisms and is capable of modifying certain parts of the design. The threat model identifies what components of the system are untrusted and what can be relied on to implement the defenses. A defense mechanism must also target specific security goals.

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 a different class 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 modification to the layout. So 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 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, and 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 assume the attacker is omniscient. Note that 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 privileged software. The SWAN solution requires access to a golden, untampered copy of the chip’s RTL description, which is used to generate a trustworthy one-time programmable fabric, and all of its related configurations. Note that we assume that all software layers, including BIOS, OS, and other privileged code, are uncompromised: an attacker who can modify those would not need to invest in devising a 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; 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, for instance with on-chip runtime checkers. In this context, an effective defense against hardware trojans must satisfy a number of key properties 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 disables the defense, this activity 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 could vary between chips. In such a scenario, because the fabric has yet to be configured at manufacture-time, the attacker will not know which physical resource will implement the attack-target design element(s), making the design of a trojan 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 point, we stated that, when the defense mechanism does detect 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 attack. 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. The SWAN’s programmable fabric is automatically generated based on the target security module selected. As mentioned earlier, only security-critical modules are mapped to this fabric, to keep overheads to a minimum.

As a case study, we examined the RISC-V BOOM out-of-order core [8] to identify modules critical to maintaining security. First, we want to ensure that the privilege rings cannot be compromised. To do 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 walker logic. Second, we must protect the execution’s control flow, so we also protect 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 contribution to chip area. Note that these modules together comprise less than 6% of the overall area of the core.

### 4.1 Secure Programmable Logic

The SWAN’s programmable fabric is optimized to increase the number of possible mappings of the target modules to the fabric, so to challenge an attacker who wish 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 logical 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. We note that, among all these components, the crossbars 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

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Consequence if compromised</th>
<th>Area (µm²)</th>
<th>% of chip area</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSRFile</td>
<td>Control status register file</td>
<td>Privilege escalation</td>
<td>24,546</td>
<td>2.62%</td>
</tr>
<tr>
<td>DecodeUnit</td>
<td>Instruction decoder</td>
<td>Execute privileged instructions, inject code</td>
<td>1,692</td>
<td>0.18%</td>
</tr>
<tr>
<td>Frontend</td>
<td>Program counter manager</td>
<td>Control flow attacks</td>
<td>3,625</td>
<td>0.39%</td>
</tr>
<tr>
<td>PTW</td>
<td>Page table walker</td>
<td>Access privileged pages</td>
<td>11,164</td>
<td>1.19%</td>
</tr>
<tr>
<td>TLB</td>
<td>Virtual to physical address translation</td>
<td>Access privileged pages, expose physical addresses</td>
<td>11,400</td>
<td>1.22%</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>52,427</td>
<td>5.59%</td>
</tr>
</tbody>
</table>
the main steps of this toolchain flow. We allow the programmer to include specific pre-processor directives in the HDL description so to guide the SWAN toolchain to use specific security extensions, discussed in the following sections. Specifically, we expect the programmer to mark especially 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. 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, the generated netlist is converted to a graph, and GraMi, an open-source graph tool [9], mines frequent subgraphs to identify good candidates to use in building the atomic fixed-function coarse-grained logic blocks used by the rest of the toolchain (step 2 in Figure 2). While a standard coarse-grained reconfigurable architecture would use large adder and multiplier blocks, our fabrics use only small clusters of gates because SWAN targets primarily control logic. To select which of the mined subgraphs will become the building blocks of the final fabric, we greedily choose those subgraphs with the most logic gates, wherever possible. Once selected, each subgraph occurrence in the netlist is replaced with a primitive logic block (undivisible cluster of gates) as shown in step 3 of the Figure. By doing so, we reduce the size of the interconnect that will be needed to support the fabric’s programmability, thus reducing area costs.

Next, the new obtained netlist is analyzed to find functionally-equivalent gates and logic blocks that are nearby in terms of their distance on the netlist graph. These blocks are grouped into sets (step 4 in Figure 2) and will provide opportunities for multiple equivalent mappings. In other words, the final manufactured fabric, all blocks in a set will be completely indistinguishable from the attacker’s perspective.

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 the structure obtained at this point 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 appropriately. 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. Note that, since each configuration is topologically identical to the original netlist, the equivalence checker’s task reduces to one of 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 multiple 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), groups containing these secure gates are made larger to ensure a higher replication of the logic blocks they include.

As an example of where camouflage can be useful, consider the processor’s control status register file. 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 only one of a small number of locations in the physical fabric, an attacker who knows this fact
We also provide a mechanism to repurpose unused logic as canaries, including an input flagged with an alarm. The programmer can access this alarm signal by modifying one canary block, its output will no longer match generated checking logic within the fabric. If an attacker puts of similar canary chains are compared via automatically generated checkers. The out-put of this register for camouflaging, our toolchain will generate a larger number of identical physical gates, all enabled to be configured into acting as the privilege register. Note that camouflaging should be used parsimoniously as it increases the size of the interconnect, which in turn translates into higher overheads as well. Thus, camouflaging is an effective knob to trade off between cost and security.

### 4.3 Canary Logic

We also provide a mechanism to repurpose unused logic as canaries for additional protection. Like their counterparts in software security, called stack canaries [10], canary logic is designed solely to detect attacks that fall on them. This serves to prevent 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 appropriate location. To prevent this lie-in-wait approach to an attack, the canaries detect when an attempted attack fails—as it is likely to do with most configurations. When canaries are to be used in 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 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, and can build a recovery system to respond to the detected attack. Using the intermediate signals in the canary checker logic further allows 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 many signals in hope of guessing the current mapping of their target, they may cause unintended errors in the process by flipping signals not related 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 allow 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 assembly house, it must exhibit different behaviors on different configurations to prove that the trusted assembly house is indeed in control of the configuration. For example, the decoder could interpret a secret opcode as an add instruction in one configuration and as a sub in another. This way, 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 logical values, resulting in different behavior among 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 and, second, security. Because different applications require different security guarantees, we provide a designer with the flexibility to build a more secure system at higher cost.

#### 5.1 Framework

We implemented our design using the RISC-V BOOM (out-of-order) core as a baseline in its small, 1-wide configuration. 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 the power and area cost into our overall results. The control status register file, decoder, front-end and PC logic, page table walker, and TLB modules...
5.2 Overheads

As a baseline, we use an ASIC implementation of the RISC-V BOOM core without SWAN, synthesized as discussed in 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 clock frequency of the full system each time.

Figure 4 shows how these attributes change as we increase the size of the SWAN fabric. Both critical path delay and power remain roughly constant across the parameterizations. Since we use a one-time programmable crossbar, timing only changes minimally since the total number of gates that the critical path runs through does not increase—a signal traverses one fuse per crossbar hop. However, the number of hops still contributes significantly to overall path delay, resulting in an increase of up to 82%. Power remains roughly constant for the same reason: since the amount of exercised logic remains constant in all configurations—one fuse per hop—and since we power gate the programming logic, the power draw of SWAN stays low. However, with more possible locations for a gate, we incur a polynomial increase in connections between cells, and thus the area cost increases rapidly due to the large size of the crossbars.

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 light-weight generality and flexibility instead of security, it is difficult for an FPGA to make the same guarantees as SWAN does with signal camouflage and canary logic. However, as a test, we implemented our target modules, with no additional canary logic, on a Xilinx UltraScale+ XCZU9EG-2FFVB1156 FPGA, which uses a 16nm technology. They consumed 1.4% of the chip’s CLB resources for an estimated 3.1mm² area cost (a 117% system-level overhead) while achieving a clock period of 12.1ns—269% of our 45nm ASIC baseline. Using estimates from [12], we approximated that this portion of the FPGA’s logic would contribute an additional 54% power overhead as well. Using the technology comparison from [13], we also estimated the area and path delay of an FPGA-based solution on a similar 45nm node. The 16nm FPGA implementation was slower and more power-hungry and costly than most of the 45nm SWAN designs—and the 45nm FPGA estimate was significantly more expensive than any parameterization of SWAN we tested.

5.3 Security Analysis

In order to properly analyze the security of the system and determine the ideal parameterization, we assume that the designers have identified all those wires inside the fabric that, if toggled by the attacker, would fail a security assertion. Furthermore, the designers must have flagged these signals with the secure directive described in Section 4.2. We assume the attacker’s goal is to flip a number of these signals. The attacker has two possible strategies: (i) they can attempt to guess the current configuration, or (ii) they can attempt to analyze the state of the fuses in the crossbars to determine the current mapping of their target gate. Since we intend SWAN to protect both security-critical logic and its checkers together, we assume that the attacker will also need to disable a checker and hide their attack to be successful.

Attacks on Random Fabric Logic. The simplest attack against SWAN is for an attacker to blindly guess the location of their target by placing a single copy on 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 is not adapted to defeat SWAN in any way, the attacker risks the trojan being exposed by accidentally triggering canary logic whenever they activate it.

For such an attack to be worthwhile for the attacker, the risk of detection must be modest in comparison with the possibility of success. However, SWAN enables the designer to control these two probabilities by adjusting the amount of signal camouflage and canaries used in the fabric. In Figure 5,
we compare a variety of possible parameterizations of SWAN to determine a Pareto optimal set of parameterizations that minimize the relative chances of a trojan succeeding versus its chance of detection given the area cost of the SWAN module. Since even a relatively low-cost parameterization of SWAN makes it significantly more likely that a trojan will be placed on a canary than on the attacker’s target and since discovery of the trojan could potentially result in exposing the actions of the malicious fab, the attacker must have a means of avoiding canaries to hope to avoid detection.

**Configuration Analysis Attacks.** In order to defeat canaries, an attacker could add a test to their simple trojan, to determine whether or not their target has been mapped to their target’s location, by checking whether a particular fuse has been set. This would ensure that the attack will only activate if it were to be successful. However, the approach will only compromise a small fraction of chips: those that were configured exactly as the attacker guessed they would be. A truly omniscient attacker could do better. By placing trojans on every location that their target could be mapped to and monitoring for when any fuse has been set that discloses the current mapping, the attacker could always know exactly where in the fabric their target resides. This attack will always succeed and will not be detected by canaries. However, it is much more expensive, because it needs to be copied across multiple locations on the fabric and needs to monitor multiple fuses for each location. This additional area cost makes it easier for the trojan to be detected with physical inspection after it is manufactured. Multiple related works reported how relatively small regions of spurious logic could be identified in a silicon chip: [7] demonstrated an imaging technique that classified individual gates at an accuracy of 85% using a 45nm node. [6] used a watermarking technique to identify trojans smaller than 90μm² on a 45nm node. This puts an upper bound on how many locations an attacker can monitor, preventing them from monitoring every location.

To determine the size of the trojan, we searched all the locations on the SWAN fabric where a block protected with signal camouflage (see Section 4.2) could possibly be mapped. Of these, we found a location where the attacker would need to monitor the fewest fuses to deduce whether their target had been mapped to the current location.

Once the trojan has analyzed the fuses, it decides whether to toggle the output of the gate it is connected to. This trojan is then copied to every other gate that the target can be mapped to, stopping when the total size of the added logic exceeds 90μm². As the number of locations a component can be placed increases, the fraction of chips where the attack succeeds will decrease. Figure 6 explores how successful the trojan will be in different parameterizations of SWAN. We show two attack scenarios: 1) the attacker analyzes as many fuses as possible with 90μm² of logic to flip a single bit in the fabric, but must also hide their actions from a checker by flipping a second bit elsewhere in the fabric, and 2) the attacker performs a minimally invasive, single point of attack by guessing blindly at the configuration, risking detection by checkers and canaries alike. If the attacker wants to flip a bit in the processor and needs to flip a second bit to disable a checker, then by ensuring that a component can be mapped to at least 8 locations, we ensure that over 99.9% of chips will not be compromised even by the omniscient attacker. Likewise, 6 possible mappings protects 99% of configurations.

**6 RELATED WORK**

In this section, we provide an overview of other defenses proposed in the literature and contrast them with our work. Note that SWAN is often orthogonal and composable with other defenses: in those cases we highlight how the related solutions complement our approach. Defenses against hardware trojans draw from three techniques: post-silicon detection, runtime detection, and obfuscation.

Post-silicon detection solutions look for trojans physically, such as with microscope inspection, power analysis, or logical analysis. TeSR (Temporal Self-Referencing) [14] aids in discovering trojans via power analysis, even when there is
no golden, trojan-free copy of the chip available. It does this by running many execution traces on the chip and looking for outliers in the expected current signature. Logical analysis often employs debugging circuitry that allows testers to rapidly explore uncommon states that the attacker could be using as a trigger signal. In [5], the circuitry is activated by a secure key, and allows for automatic exploration of possible states, with traces packaged and logged on the fly, so that if the trojan is activated, its payload is observed immediately. 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. However, by allowing different configurations of SWAN to manifest different functionalities, simple logical tests expose whether or not the attacker has interfered with the configuration process.

Runtime detection includes all kinds of on-chip and off-chip checking logic that monitor for violations of the security rules of the chip. [15] proposed a method to generate such checkers directly from the specification, ensuring high coverage of potential attack surfaces. [16] explores the possibility that the runtime checker may have been itself been tampered with. 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 with which checkers can be protected and guaranteed observability into the components they protect. SAWN 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 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 the gates in the design 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 is intractable when many LUTs are present. A complete FPGA-like reconfigurable fabric has also been proposed as a form of obfuscation. In [4], the authors use an FPGA fabric to implement instruction set randomization on a processor to 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 blocks in the design, it also hides their physical locations as well.

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 their design needs. 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—thus SWAN is still significantly cheaper than an FPGA implementation, which would at least double the area while providing far more limited security guarantees.

ACKNOWLEDGEMENTS

This work was supported in part by DARPA grant #HQ007271724100.

REFERENCES