Private Circuits III: Hardware Trojan-Resilience via Testing Amplification

Stefan Dziembowski
University of Warsaw
Institute of Informatics
stefan@dziembowski.net

Sebastian Faust
University of Bochum
Fakultät für Mathematik
sebastian.faust@gmail.com

François-Xavier Standaert
Université catholique de Louvain
ICTEAM – Crypto Group
fstandae@uclouvain.be

ABSTRACT

Security against hardware trojans is currently becoming an essential ingredient to ensure trust in information systems. A variety of solutions have been introduced to reach this goal, ranging from reactive (i.e., detection-based) to preventive (i.e., trying to make the insertion of a trojan more difficult for the adversary). In this paper, we show how testing (which is a typical detection tool) can be used to state concrete security guarantees for preventive approaches to trojan-resilience. For this purpose, we build on and formalize two important previous works which introduced "input scrambling" and "split manufacturing" as countermeasures to hardware trojans. Using these ingredients, we present a generic compiler that can transform any circuit into a trojan-resilient one, for which we can state quantitative security guarantees on the number of correct executions of the circuit thanks to a new tool denoted as "testing amplification". Compared to previous works, our threat model covers an extended range of hardware trojans while we stick with the goal of minimizing the number of honest elements in our transformed circuits. Since transformed circuits essentially correspond to redundant multiparty computations of the target functionality, they also allow reasonably efficient implementations, which can be further optimized if specialized to certain cryptographic primitives and security goals.

1. INTRODUCTION

While modern cryptography generally assumes adversaries with black box access to their target primitives, the last two decades have witnessed the emergence of increasingly powerful physical attacks that circumvent this abstract model. Side-channel analysis [23, 24] and fault attacks [10, 9] are typical examples of such concerns, where the adversary can respectively observe physical leakages produced by an implementation, or force it to perform erroneous computations. In this respect, hardware trojan attacks can be viewed as the ultimate physical attack, where the adversary can even modify the implementations at design time, in order to hide a backdoor that may be used after deployment. This threat has recently gained attention, since the increasing complexity of electronic systems, and the ongoing trend of outsourcing chip fabrication to a few specialized foundries, has made it more and more realistic, with possibly catastrophic consequences for security and safety [2, 8]. As documented in [7, 28] the attacks by a malicious manufacturer are also hard to prevent, since they can lead to very diverse attack vectors, with various activation mechanisms and payloads.

In this context, and looking back at the already broad literature on countermeasures against side-channel and fault attacks, an important lesson learned is that the most effective protections usually rely on a good separation of duties between well-chosen (generally physical) assumptions and sound mathematical principles to amplify them. Taking one emblematic example, masking improves security against side-channel attacks by relying on the assumption that physical leakages are noisy, and by amplifiying this noise thanks to secret sharing [13]. Based on the similar (physical) nature of hardware trojans, it is therefore reasonable to expect that solutions to prevent them may follow a similar path. In this respect, and starting at the hardware level, detection thanks to side-channels possibly amplified by some fingerprinting has been studied, e.g., in [1, 3, 26]. Very summarized, such approaches are powerful in the sense that they are in principle able to detect any type of trojan, including purely physical ones (e.g., triggered by a temperature change and sending secret messages through an undocumented antenna), which makes them an important part of the puzzle. But they are also inherently limited by their heuristic nature and sometimes strong assumptions. For example, they work best in presence of a golden (trusted) chip that may not be easily available, and the effectiveness of the detection decreases when reducing the size of the trojan circuitry.

In this paper, we therefore tackle the question whether more formal solutions can help to rule out well-defined classes of hardware trojan attacks, and achieve stronger resistance in practice. For this purpose, we build on two important previous works. In the first one, Waksman and Sethumadhavan consider digitally triggered trojan attacks [30]. A digitally triggered trojan is a malicious piece of hardware that delivers its payload when a digital input is given to the device. This can be done, for instance, through so-called "cheat codes" or "time bombs". The first type of attack triggers the malicious behavior of the trojan when a certain input is provided to the device, while "time bombs" activate the trojan, e.g., after the device is executed for a certain number of times. The work of Waksman and Sethumadhavan provides ad-hoc countermeasures against these two types of

CCS'16, October 24 - 28, 2016, Vienna, Austria
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4139-4/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2976749.2978419

attacks. In particular, they propose to scramble the inputs to defeat the cheat codes and use power resets to protect against (volatile) time bombs. In the second one, Imeson et al. introduce the concept of "split manufacturing" for obfuscation, as a way to make it hard for an adversary to identify the gates of an implementation that he would need to modify to mount his attack [20]. The main contribution of our work is to provide *generic countermeasures* for significantly *broader classes of trojan attacks*, and to provide a *formal framework* in which these countermeasures can be analyzed and concrete security bounds can be derived. We describe our technical contribution in more detail below.

Types of hardware trojans. Similar to Waksman and Sethumadhavan, we consider a setting where the production of a device is outsourced to a potentially malicious hardware manufacturer. The manufacturer produces a set of devices $\mathsf{D}_1, \ldots, \mathsf{D}_\ell$ that supposedly implement functionalities $\Gamma_1, \ldots, \Gamma_\ell$, but may contain trojans and react maliciously. As in [30] we restrict the type of malicious behavior to hardware trojans that are digitally triggered, such as cheat codes or time bombs. Besides formally modeling such digitally triggered trojans, we also extend the model of Waksman and Sethumadhavan by not only considering volatile time bombs (i.e., where the clock needs to be powered) but also non-volatile ones (which may become hard to detect in highly integrated electronic systems).

The trojan protection schemes. To protect against digitally triggered hardware trojans we introduce so-called *trojan protection schemes.* A trojan protection scheme consists of two components: a circuit transformation TR and a tester T. The transformation describes a method to compile an arbitrary functionality described as an arithmetic circuit Γ into a protected specification consisting of a trusted master circuit M and a set of circuits $\Gamma_1, \ldots, \Gamma_\ell$. We assume that M has to be implemented in a trusted way and its production is not outsourced to the malicious hardware manufacturer $\mathcal{A}$, while the devices D_i are produced by $\mathcal{A}$. To obtain a stronger result, we require that M is as simple as possible. For our concrete construction M will consist of a couple of wires and a small number of simple gates – in particular, the size (counted as the number of gates) of M is independent of the size of Γ. The implementation of our transformed circuits therefore follows the same "split manufacturing" principles as introduced by Imeson et al. [20]. The second component of the trojan protection scheme is a tester T. The tester verifies if the devices D_i correctly implement the functionality Γ_i. Such tests typically involve whether the input/output behavior of D_i corresponds to the input/output behavior of the honest specification Γ_i.

Robustness of trojan protection schemes. The main security guarantee that our trojan protection scheme shall achieve is called *robustness.* Informally, robustness is modeled by a game with two phases. First, in the testing phase the tester T checks whether the devices D_i implement the corresponding specification Γ_i. If the testing is passed the adversary can in a second phase interact with the device composed of the trusted master M and the devices D_i. Robustness guarantees that for the same inputs, the outputs produced in the second phase by the device are identical to the outputs produced by the honest specification Γ. Robustness is parameterized by two parameters t and n, where t denotes the number of tests carried out by T and n is the number of executions for which the output produced by the device has to be identical to the honest specification Γ. Typically, for our constructions we require $t > n$.

A trojan protection scheme for any functionality Γ. Our main contribution is the design of a trojan protection scheme that achieves robustness for any functionality Γ. We next give a high-level description of our trojan protection scheme omitting several technical details.

As a first step, the transformation compiles the specification Γ into three so-called *mini-circuits* $(\Gamma^0, \Gamma^1, \Gamma^2)$. These mini-circuits emulate Γ using a passively secure 3-party protocol, where the inputs to $(\Gamma^0, \Gamma^1, \Gamma^2)$ are secret-shared by the trusted master circuit M.

The first observation in order to achieve robustness is that if the mini-circuits Γ^i follow exactly the secure 3-party protocol, then they do not learn anything about the user provided input. Hence, a malicious user is hindered in activating the trojan by choosing a special input. Of course, once Γ^i gets implemented by $\mathcal{A}$ nothing stops $\mathcal{A}$ to produce devices D^i that do not follow the protocol, e.g., by transmitting their shares to the other devices. Such a behavior will, however, be detected with good probability during the testing phase.

The above only prevents activation of the trojan by a malicious input, but does not deal yet with an activation via time-bombs. For instance, assume that the trojan is activated only after the $(t+1)$-th execution. If we test devices only for t times, the malicious behavior will not be detected and achieving robustness is infeasible. To circumvent this, we randomize the number of tests t', where t' is drawn uniformly from $\{1, \ldots, t\}$. Since (i) the total number of executions after testing is bounded by n and (ii) test and real executions look the same from the device's point of view (due to the 3-party computation), we can bound the probability that malicious behavior is triggered by time bombs.

Unfortunately, the above gives only a weak security bound. It is, however, easy to amplify security by letting $\mathcal{A}$ produce λ independent copies $(\mathsf{D}_1^0, \mathsf{D}_1^1, \mathsf{D}_1^2), \ldots, (\mathsf{D}_\lambda^0, \mathsf{D}_\lambda^1, \mathsf{D}_\lambda^2)$, so that $\ell = 3\lambda$, where each such tuple is tested for a random and independent number of times t_i. In our final construction, the master M then runs each of the tuples on independent input sharings and takes the majority of the results to produce the final output with good robustness. Concretely, we can guarantee correct execution with probability $\left(\frac{n}{t}\right)^{\lambda/2}$.

Applications of trojan protection schemes. The requirement that $t > n$ can be viewed as a limitation of our work, but we argue in Section 3 that this condition is in fact necessary for testing-based security against hardware trojans. Hence, such schemes are only applicable in settings where there is an a-priori bound on the number of times the device is used. Such bounded number of executions naturally occurs when a user manually interacts with a device. Since testing can be automatized it is then feasible to carry out millions of test cases, while after deployment many devices are used only a few thousand times. There are many examples of such settings (e.g., opening doors). Their relevance naturally increases with the sensitivity of the data to protect and not too limited cost constraints, such as electronics in planes used for starting and landing (which have natural restrictions on the number of executions). We stress that nothing prevents the master circuit M to count the number of runs, issue a warning when the executions limit is reached, and then to re-perform a testing phase.

Implementation issues. Despite our primary focus is on the genericity and formal guarantees of the proposed countermeasure against hardware trojans, we also take implementation issues into account in our developments. In this respect, we first limit the use of trusted components to some routing and a couple of gates, as in [20]. We argue in Section 3 why a minimum complexity (i.e., the presence of gates in M) is necessary for testing-based security against hardware trojans. Next, and as far as performances are concerned, the main efficiency bottleneck in our trojan protection scheme is the use of a passively secure 3-party computation protocol. We discuss the time and area overheads it implies for a mainstream cryptographic functionality such as a standard block cipher in Section 5.2. And we conclude the paper by showing that better efficiency can be achieved if we aim for protecting specific cryptographic primitives. In particular, we give constructions for a PRG and a MAC that only increase the complexity by a linear factor ℓ (compared to the unprotected scheme), while guaranteeing security except with probability $O(2^{-\ell})$. Notice that these schemes also have two additional benefits compared to our generic solution: first, except for an initial secret sharing of the inputs their execution does not require the sometimes costly generation of pseudorandomness, and second they require almost no interaction between the sub-devices.

Related works. In a separate line of papers, Bayer and Seifert [6] and very recently Wahby et al. and Ateniese et al. [29, 5] consider a setting where an untrusted ASIC proves, each time it performs a computation, that the execution is correct. These papers build on a large literature on verifiable computation, probabilistically checkable proofs and other, related topics. Compared to our work, such approaches and techniques correspond to a different tradeoff between security and trust. On the one hand, they cover even broader classes of hardware Trojans and achieve security for an arbitrary number of executions (unlike us who restrict the number of executions a-priori). On the other hand, they require a trusted verifier which is typically more complex than our master circuit that does only routing and and uses a small number of gates.[1] These works also aim at different goals than ours. Namely, whenever a proof of correct execution is not verified in [29, 5], the system stops. By contrast, we can guarantee a number of correct executions and therefore are also resistant against denial-of-service attacks.

The work of Haider et al. [19] also shares similarities with ours, and provides a formal analysis of trojan detection using pre-silicon logic testing tools.

Eventually, our constructions follow the seminal investigations of Ishai et al. who introduced circuit transformation in the field of side-channel and fault attacks [21, 22]. Their results on "Private Circuits" (I and II) motivated us to look at generic compilers for trojan-resilient circuits, which is a natural next step in the study of physical adversaries against cryptographic hardware. Conceptually, the principles we exploit in our trojan protection schemes are also close to masking against side-channel attacks. Namely, masking exploits secret sharing and multiparty computation in order to amplify the impact of noise in leaking cryptographic implementations. Similarly, we exploit these techniques to amplify the impact of testing against hardware trojans.

[1]Comparing the efficiencies of prover sides is more challenging since application-dependent, and is therefore an interesting scope for further research.

2. TROJAN PROTECTION SCHEMES

2.1 The model of computation

2.1.1 The circuit specification

Computation carried out by an algorithm is abstractly defined via a *specification*. We model the specification as a circuit Γ, which is represented by a Directed Acyclic Graph (DAG). The set of vertices of the graph represents the gates of the circuit, while the edges are the wires connecting the gates. The wires in the circuit carry elements from a finite field $\mathbb{F}$, while the gates carry out the operations in the finite field or take special task such as storing values. The simplest case is when $\mathbb{F}$ is the binary field, in which case the wires carry bits, and the gates, for instance, represent the Boolean operations AND (next denoted by $\odot$) and XOR (next denoted by $\oplus$). For simplicity, all the gates are assumed to have at most fan-in two. On the other hand, gates may have arbitrary fan-out, where we assume that all output wires carry the same value. We will also consider the arithmetic circuits, where $\mathbb{F}$ is a larger field, and the gates represent the corresponding arithmetic operations.

In addition to the standard Boolean/arithmetic gates, we allow the specification Γ to contain two additional gates: the randomness gates rand and (volatile and non-volatile) memory gates. The randomness gate has no incoming wires but can have an arbitrary number of outgoing wires, which carry a random element from $\mathbb{F}$. One may think of rand as a gate producing true randomness. Next, the non-volatile memory gates (next called registers), are used to store the results of the computation's different (clock) cycles and only maintain their state when the chip is powered. Registers have a single incoming wire and an arbitrary number of outgoing wires. They can be placed everywhere in the circuit, but we require that each cycle of Γ contains at least one register. Eventually, non-volatile memory gates (next called memory gates for short) play a similar role as volatile ones, but maintain their state even if the chip is not powered.

To complete the description of the circuit, we also need to explain how it processes inputs/outputs. The circuit takes inputs $\vec{x} \in \mathbb{F}^{\alpha}$ and the outputs $\vec{y} \in \mathbb{F}^{\beta}$ as a result of the computation are delivered to the user. One may view them as wires carrying the inputs and outputs, respectively, that are connected to the "outside world" of the circuit. For a circuit that takes as input $\vec{x}$ and produces output $\vec{y}$ we write $\vec{x} \leftarrow \Gamma(\vec{y})$ and call it a *run* of Γ or a *round*, which usually takes several clock cycles to be executed.

Beside the public inputs/outputs, the circuit also may keep a (secret) state between runs of Γ. The secret state of Γ is initially set through the Init operation and kept in non-volatile memory gates. We write $\mathsf{Init}(\Gamma, \vec{m})$ when the initial state of Γ is set to $\vec{m}$. Notice that the state of Γ may change via the public inputs $\vec{x}$, in which case we say Γ is *stateful*. Otherwise, if the state is only written once via the Init procedure, we say that Γ is *stateless*. For a computation on input $\vec{x}$ and an initial state $\vec{m} \in \mathbb{F}^{s}$, we write $\vec{x} \leftarrow \Gamma[\vec{m}](\vec{y})$. If Γ has been run for many rounds then $\vec{m}$ may already have been changed.

2.1.2 Circuit compilers

The goal of the circuit compiler (or transformer) $\mathsf{TR} = (\mathsf{TR}_1, \mathsf{TR}_2)$ is to compile a specification described as a circuit Γ into a protected specification Γ'. We write for the compilation process: $\Gamma' \leftarrow \mathsf{TR}_1(1^k, \ell, \Gamma)$, where k is the computational security parameter (e.g., used for the PRG in our following generic construction) and $\vec{m}' \leftarrow \mathsf{TR}_2(1^k, \ell, \vec{m})$ for compiling the initial state. In the following, we will often abuse notation and omit to explicitly denote the compiled state by $\vec{m}'$ since in our construction it will be just a longer (secret-shared) vector. Also, we will sometimes omit to mention explicitly the parameter ℓ, when it is clear from the context. The specification of Γ' consists of two parts. First, a set of sub-circuits $\Gamma_1, \ldots, \Gamma_\ell$ and second, a so-called *master circuit* M.[2] The role of the master circuit M is to manage the communication between the sub-circuits Γ_i and the user of these circuits. While these sub-circuits can be constructed using the above described gates, for ease of notation we choose to describe them with an additional feature for communication. That is, we allow the sub-circuits Γ_i to communicate with M and vice versa. To this end, we introduce commands having the form $(\mathsf{cmd}, \mathsf{val})$, where cmd is a label denoting the command that shall be executed and val is an accompanied element in $\mathbb{F}$ (or a vector thereof). We will consider the following types of commands:

1. The command $((\mathsf{send}, j), \vec{x})$ is sent by Γ_i to M to specify that Γ_i wants to send message $\vec{x}$ to the circuit j.
2. The command $(\mathsf{in}, \vec{x})$ is sent by M to Γ_i to specify that Γ_i receives message $\vec{x}$ as input.
3. The command $(\mathsf{out}, \vec{y})$ is sent by Γ_i to M to specify that Γ_i's output is ready and worth $\vec{y}$.

The evaluation of the sub-circuits $\Gamma_1, \ldots, \Gamma_\ell$ with master M on input $\vec{x}$ with initial state $\vec{m}$ producing output $\vec{y}$ will next be written as $\vec{y} \leftarrow (\mathsf{M} \Leftrightarrow \Gamma_1, \ldots, \Gamma_\ell)[\vec{m}](\vec{x})$, where $\vec{m}$ is the initial compiled state. One may think of $(\mathsf{M} \Leftrightarrow \Gamma_1, \ldots, \Gamma_\ell)$ as a circuit composed of the sub-circuits Γ_i and M, where the composition is specified by the communication commands between Γ_i and M.

In the following we will often need to describe the *view* of a circuit Γ_i. The view of Γ_i includes all command/value pairs denoted by $(\mathsf{cmd}, \mathsf{val})$ that Γ_i receives/sends from/to M. The view of Γ_i is denoted by $\mathsf{View}(\Gamma_i[\vec{m}](\vec{x}))$ and contains tuples of the form $(\mathsf{cmd}, \mathsf{val})$. Notice that the view also includes the inputs/outputs given by M to Γ_i.

The simplest property that we require from the transformation is correctness. That is, for all $\vec{m}, \vec{x}_i$ it holds that outputs produced by Γ on initial state $\vec{m}$ with input $\vec{x}_i$ are identical to the outputs produced by Γ' on initial state $\vec{m}'$ with input $\vec{x}_i$. Notice that if Γ was a randomized circuit (i.e., it uses rand gates), then we require that the output distributions are computationally indistinguishable.

The second property that we require is robustness against malicious manufacturers, which we introduce in the next section. To look ahead, we will typically let the manufacturer produce devices D_i that take the role of Γ_i, while the master M is required to be implemented honestly. Of course, due to this assumption, the latter has to be as simple as possible (in our case typically M will only require wiring devices together and a few very basic operations).

2.2 Security against malicious manufacturers

Consider a circuit specification Γ with initial state $\vec{m}$ and let $(\Gamma', \vec{m}') \leftarrow (\mathsf{TR}_1(1^k, \ell, \Gamma), \mathsf{TR}_2(1^k, \ell, \vec{m}))$, where $\Gamma' = (\mathsf{M}, (\Gamma_1, \ldots, \Gamma_\ell))$. We are interested in a setting where a potential malicious manufacturer $\mathcal{A}$ gets as input the specifications $(\Gamma_1, \ldots, \Gamma_\ell)$ and produces a set of devices $\mathsf{D}_1, \ldots \mathsf{D}_\ell$, where D_i supposedly implements some functionality Γ_i. A device D_i takes some input $\vec{x}$ and produces an output $\vec{y}$. In order to compute $\vec{y}$ from the input $\vec{x}$ the device D_i may communicate with the master circuit M, which is implemented honestly. To this end, it can send and receive commands of the form $(\mathsf{cmd}, \mathsf{val})$ to/from M. While the devices D_i can in principle implement any functionality (since they are built by the malicious hardware manufacturer), we require that an implementation of D_i can be simulated using our circuit model above, as formalized by the following assumption.

Assumption 1. *Let D_i be the devices output by $\mathcal{A}$. We require that there exists (possibly probabilistic) circuit specifications $\widetilde{\Gamma}_i$ such that for all public inputs $\vec{x} \in \mathbb{F}^\alpha$ and any initial state $\vec{m} \in \mathbb{F}^s$, we have* $\mathsf{View}(\mathsf{D}_i[\vec{m}](\vec{x})) \equiv \mathsf{View}(\widetilde{\Gamma}_i[\vec{m}](\vec{x}))$.

Informally, the assumption says that as long as a trojan attack can be modeled by a (possibly probabilistic) circuit, then the attack is within our security model. Note that this allows for fairly general trojan attacks. For instance, we will not make any assumption of the computational complexity of the trojan other than it was produced by a PPT adversary $\mathcal{A}$. This, e.g., means that it can be more complex than the computation carried out by the honest specification Γ_i. We note also that some restriction on the power of the trojan attack is *necessary.* For instance, if the trojan embeds an antenna into the device that sends secret data via a side-channel to the attacker, then security is hard to achieve. Looking ahead, at a technical level Assumption 1 is also crucial for the security proof and shows up in Theorem 1.[3] We discuss the plausibility of Assumption 1 and the attacks that are (not) incorporated in our model in Section 5.4.

2.2.1 Testing

Once the devices D_i have been produced by the (malicious) manufacturer, they are tested by a PPT tester T. The goal of T is to verify whether each D_i implements its corresponding functionality given by the circuit specification Γ_i. We consider black-box testing. That is, T can specify the inputs of D_i and communicate with D_i over the specified interface. To this end, the tester will typically take the role of the master M, i.e., the tester can run the manufactured devices D_i on chosen inputs and verify whether the results correspond to the results produced by the honest functionality Γ_i. Notice that these tests typically also include the verification of the communication with M. We will write $b \leftarrow \mathsf{T}^{\mathsf{D}_1(\cdot), \ldots, \mathsf{D}_\ell(\cdot)}(1^k, \Gamma)$, where T can interact with the devices D_i via the communication commands and at the end of the test outputs a bit b indicating whether the test has passed or failed. We call the tester T t-bounded, if each of the D_i is run for at most t times.

2.2.2 Trojan protection schemes

A trojan protection scheme $\Pi := (\mathsf{TR}, \mathsf{T})$ consists of the circuit transformation TR and the testing algorithm T. We

[2]Notice that $\Gamma_1, \ldots, \Gamma_\ell$ and M are described using the circuit model from above and therefore may include memory cells.

[3]Concretely, in our construction the devices D^i supposedly run a passively secure 3-party protocol. At some point in the proof we want to replace the physical devices D^i by some abstract description of a circuit $\tilde{\Gamma}^i$ (which is not necessarily the same as Γ^i) that emulates the malicious behavior of D^i. At this point in the proof we need Assumption 1.

```
Game $\mathsf{ROB}_\Pi(\mathcal{A}, \mathsf{pub}, \Gamma, \vec{m})$:
$((\mathsf{M}, \{\Gamma_i\}_i), \vec{m}') \leftarrow (\mathsf{TR}_1(1^k, \ell, \Gamma), \mathsf{TR}_2(1^k, \ell, \vec{m}))$
$\{\mathsf{D}_i\}_i \leftarrow \mathcal{A}(1^k, (\mathsf{M}, \{\Gamma_i\}_i))$
Set the initial state of the devices $\mathsf{Init}(\{\mathsf{D}_i\}_i, \vec{m}')$
If $\mathsf{T}^{\mathsf{D}_1(.),\ldots,\mathsf{D}_\ell(.)}(1^k, (\mathsf{M}, \{\Gamma_i\}_i)) = \mathsf{false}$ then return 0
$\vec{x}_1 \leftarrow \mathcal{A}(1^k)$
For $i = 1$ to $n$ repeat:
    $\vec{z}_i \leftarrow (\mathsf{M} \Leftrightarrow \mathsf{D}_1, \ldots, \mathsf{D}_\ell)[\vec{m}'](\vec{x}_i)$
    $\vec{y}_i \leftarrow \Gamma[\vec{m}](\vec{x}_i)$
    If $\vec{y}_i \neq \vec{z}_i$ then return 1
    $\vec{x}_{i+1} \leftarrow \mathcal{A}(1^k, \vec{z}_i)$
Return 0.
```

Figure 1: The robustness game ROB_Π.

model security of the trojan protection scheme Π against a malicious manufacturer by a robustness game denoted by ROB_Π, given in Figure 1. In the game, we first run the transformation to obtain the specification of the protected circuit $((\mathsf{M}, \{\Gamma_i\}_i), \vec{m})$. Next, the specification is given to the malicious manufacturer $\mathcal{A}$ who outputs a set of devices $\{\mathsf{D}_i\}_i$. The devices are tested by T, and if the testing succeeds then $\mathcal{A}$ may interact with $\vec{z}_i \leftarrow (\mathsf{M} \Leftrightarrow \mathsf{D}_1, \ldots, \mathsf{D}_\ell)[\vec{m}](\vec{x}_i)$ by specifying an input $\vec{x}_i$ and receiving the output $\vec{z}_i$. We say that $\mathcal{A}$ *wins* the game iff after the testing has succeeded, he manages to produce an output $\vec{z}_i$ that differs from the output $\vec{y}_i$ of a correct computation on input $\vec{x}_i$, i.e., $\vec{y}_i \leftarrow \Gamma[\vec{m}](\vec{x}_i)$. Note that for our constructions, we will require that the number of tests t done by T is larger than the number of executions n. We state the security properties of a trojan protection scheme as:

Definition 1. Let ℓ, n, t, and k be some natural parameters. A trojan protection scheme $\Pi = (\mathsf{TR}, \mathsf{T})$ is (t, n, ϵ)-trojan robust if the following two conditions hold:

1. The tester T is t-bounded,
2. For any manufacturer $\mathcal{A}$, any circuit Γ and any initial state $\vec{m}$ we have:

   $$\Pr[\mathsf{ROB}_\Pi(\mathcal{A}, \ell, n, t, k, \Gamma, \vec{m}) = 1] \leq \epsilon,$$

   where the probability is taken over the internal coin tosses of $\mathcal{A}$ and the coin tosses of the game ROB_Π.

To simplify the notation in the sequel we will use a symbol pub as a shorthand for the tuple consisting of "public parameters" in ROB, i.e. we will set $\mathsf{pub} := (\ell, n, t, k)$, and write $\mathsf{ROB}_\Pi(\mathcal{A}, \mathsf{pub}, \Gamma, \vec{m})$.

3. IMPOSSIBILITY RESULTS

We now discuss some inherent limitations of the testing techniques presented in the previous section. First, we argue that in most of the realistic applications the maximal number t of testing rounds should be much larger than the number n of times that the device will be used. For this purpose, consider a single device D and suppose that the malicious manufacturer designed it in such a way that it behaves as its specification requires, except with probability ϵ (whose value we will determine later). More precisely let Bad_i denote the event that in the ith round of its life (during testing or the real execution) D behaves wrongly (for example: it starts to produce wrong results, or it terminates). Assume that the Bad_i's are independent, and $\Pr(\mathsf{Bad}_i) = \epsilon$.

The probability that this malicious actions are not detected during testing is equal to $\Pr(\neg\mathsf{Bad}_1 \wedge \cdots \wedge \neg\mathsf{Bad}_{t_0})$, where $t_0 \leq t$ is the number of rounds of test. This, clearly, is at at least equal to $(1-\epsilon)^t$. Similarly the probability that a Bad event happened during one of the n rounds of execution is equal to $1-(1-\epsilon)^n$. Hence the probability p that D passed the tests and failed during the execution is at least equal to $(1-\epsilon)^t \cdot (1-(1-\epsilon)^n)$. Now suppose that the adversary sets $\epsilon := 1-(t/(n+t))^{1/n}$. Then p is at least $(t/(n+t))^{t/n} \cdot n/(n+t) = (1+n/t)^{-t/n} \cdot n/(n+t)$, which is at least equal to $n/(e \cdot (n+t))$, where e is the base of the natural logarithm (this is because $(1+n/t)^{-t/n} \geq e^{-1}$).

This in particular means that if t is small then with very good probability (at least $n/(e \cdot (n+t))$) the adversary's device behaves correctly during the testing, and incorrectly during the real-life execution. This shows that in reality we will usually need to have $t \gg n$ if we want to get high assurance that the device will not fail during the execution. Also, since this probability is inversely proportional to the number t of tests, thus, intuitively, to obtain error probability smaller than $O(n^{-c})$ (for some c) we need to have at least c devices D in the system.

This last statement is of course informal, since in order to formalize it, we would need to restrict the power of the master circuit (in principle every computation can be done in a perfectly secure way if it is performed by the trusted master). For this purpose, we next state some simple observations regarding the necessary complexity of the master circuit. First, note that the above observations imply that, in order to get any security beyond the "$n/(e \cdot (n+t))$" barrier, none of the D_i gadgets can be "directly connected to the output", i.e., the master circuit M cannot just forward the outputs from D_i as its own output (without performing any computation on this value). This is because the adversary can make such "unprocessed" output to be wrong with probability $n/(e \cdot (n+t))$. It justifies why we always need some kind of "output processing" (which, in our case, will be handled by a majority gate).

A similar fact can be shown about the input processing, i.e., we can prove that in most of the cases no M can pass its input directly to one of the D_i gadgets. Observe, that the above fact certainly cannot hold for *all* functionalities Γ. For illustration, suppose that Γ ignores its input (e.g., it is a pseudorandom generator whose output depends only in the initial state and does not depend on the inputs). Then it can be implemented by $(\mathsf{M} \Leftrightarrow \mathsf{D}_1, \ldots, \mathsf{D}_\ell)$ such that M sends its inputs directly to the some "dummy" D_i's that do not perform any actions. To be more formal, let us say that a circuit Γ is *simple* if it contains no gates (i.e. it has only wires). We say that a circuit Γ *can be simplified* if for every initial state $\vec{m}$ there exists a sequence $\{\Gamma_i\}_{i=1}$ of simple circuits which, for every sequence $\{\vec{x}_i\}_i$ of inputs Γ with initial state $\vec{m}$ and rounds inputs $\{\vec{x}_i\}_i$, produces the same output as $\{\Gamma_i\}_{i=1}$ on inputs $\{\vec{x}_i\}_i$ (where in round i we apply Γ_i to $\vec{x}_i$). Intuitively, a circuit, *cannot* be simplified if it performs some non-trivial operations on its input.

We now show that every such a circuit Γ cannot be simulated by a circuit $\Gamma' = (\mathsf{M} \Leftrightarrow \mathsf{D}_1, \ldots, \mathsf{D}_\ell)$, where M is simple (and in particular, every circuit with simple M can be broken with probability close to 1 for n that does not depend on t). We consider circuits Γ that do not have any randomness gates, but our argument can be generalized also to the case of circuits with random gates.

Lemma 1. *Consider a trojan protection scheme* $\Pi = (\mathsf{TR}, \mathsf{T})$*. Suppose it produces as output only circuits* $(\mathsf{M}, \{\Gamma_i\}_i)$ *such that* M *is simple. Let* Γ *be a circuit that cannot be simplified, and suppose* $\ell(k)$ *is the number of sub-circuits* Γ_i *that* Π *produces on input* $(\Gamma, 1^k)$*. Then the scheme* Π *is not* (t, n, ϵ)*-trojan robust for any* $t, k, n = (\ell(k)+1)\cdot k$*, and* $\epsilon < 1 - (\ell(k)\cdot t + 1)\cdot |\mathbb{F}|^{-k}$.

The proof of the lemma appears in the extended version of this paper [15]. Summarizing, the above statements highlight that the complexity of both the testing phase and the master circuits in the following constructions (measured in number of tests and gates) is essentially necessary.

4. TROJAN RESILIENT CIRCUITS

To simplify our analyses, we first consider the case when Γ is deterministic and does not update its initial state. This means that once the state has been initialized to $\vec{m}$ it is never changed by the computation of Γ. AES implementations are an example of such circuits. In Section 4.5 we then discuss how to extend our results to circuits that update their state (e.g., stream cipher) and are probabilistic.

4.1 Our basic construction

The compiler TR takes as input a description of a (binary/arithmetic) circuit Γ and outputs λ *sub-circuits* $\Gamma_i := (\Gamma_i^0, \Gamma_i^1, \Gamma_i^2)$ for $i \in [\lambda]$ and the master circuit M. Each subcircuit consist of three *mini-circuits* so that $\ell = 3\lambda$. While the λ sub-circuits operate independently from each other (i.e., there is no communication between them), the mini-circuits of each sub-circuit are connected through M.

The processing of an input $\vec{x} \in \mathbb{F}^\alpha$ with an initial secret input $\vec{m}$ resulting in an output $\vec{y} \in \mathbb{F}^\beta$ proceeds in three phases: (i) the input pre-processing phase, (ii) the computation phase and (iii) the output post-processing phase. The bulk of the computation is carried out in phase (ii), while phase (i) and (iii) are carried out by the master M. Since the implementation of M has to be trustworthy, we will minimize the work of M. In particular, we require that the number of gates (but not the number of wires) used by M is *independent* of the number of gates used by Γ; instead, it will depend only on Γ's input size α and output size β. The overall structure of the specification of the transformed circuit Γ' is given in Figure 2.

In the *pre-processing phase* on input $\vec{x}$ the master circuit M produces λ additive 2-out-of-2 secret sharings of $\vec{x}$. More formally, it proceeds as follows:

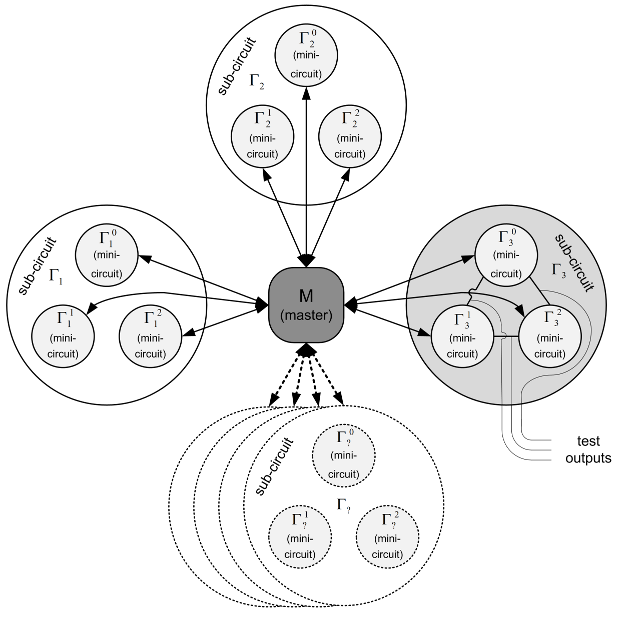
1. Repeat the following for $i \in [\lambda]$:
   (a) Sample $\vec{r}_i \leftarrow \mathbb{F}^\alpha$ using α rand gates.
   (b) Compute $\vec{s}_i = \vec{x} - \vec{r}_i$.
2. Output $\{(\vec{r}_i, \vec{s}_i)\}_i$.

Note that such a pre-processing does not imply that the master cricuit needs to generate trusted randomness. As discussed in Section 6.1, we can use an efficient construction of trojan-secure PRG for this purpose.

In the *computation phase*, each triplet of sub-circuits $\Gamma_i := (\Gamma_i^0, \Gamma_i^1, \Gamma_i^2)$ implements computation of the circuit Γ using a passively secure 3-party protocol. While in principle *any* construction of a passively secure 3-party protocol will work, we chose to present a particular protocol which is well-suited for our application and allows efficient hardware implementations.[4] In our construction the λ sub-circuits Γ_i carry out exactly the same computation, where Γ_i uses the public input tuple $(\vec{r}_i, \vec{s}_i)$. Since the computation of each sub-circuit is identical, to ease notation, in the following we omit to explicitly mention the index i. The triplet $(\Gamma^0, \Gamma^1, \Gamma^2)$ evaluates the circuit Γ gate-by-gate. That is, each gate in Γ is processed by the sub-circuit $(\Gamma_0, \Gamma_1, \Gamma_2)$ running a secure 3-party protocol emulating the operation of the gate in Γ. In the computation phase, the role of the master M is restricted to forward commands between mini-circuits. In particular, to initiate the computation of $(\Gamma^0, \Gamma^1, \Gamma^2)$ the master M sends the following command to Γ^i:

1. $(\mathsf{in}, \vec{r})$ to Γ^1 and $(\mathsf{in}, \vec{s})$ to Γ^2, respectively.
2. $(\mathsf{in}, \varnothing)$ to Γ^0. Notice that this means that Γ^0 is independent of the inputs of the computation.

On receiving the in command, the mini-circuits $(\Gamma^0, \Gamma^1, \Gamma^2)$ will then run one of the protocols shown in Figure 3 depending on the type of gates in Γ. The basic invariant is that $(\Gamma^0, \Gamma^1, \Gamma^2)$ guarantee that for a gate g in Γ that outputs c, we have that at the end of the protocol Γ^1 produces c_1 while Γ^2 computes c_2 such that (c_1, c_2) represents a random sharing of c. In other words: each value on a wire in Γ is shared between Γ^1 and Γ^2. The mini-circuit Γ^0 is involved only for computing the field multiplication by providing correlated randomness. To generate randomness, Γ^0 will use an implementation of a secure pseudorandom generator $\mathsf{prg} : \mathbb{F}^s \rightarrow \mathbb{F}^\kappa$.[5] To this, end it holds an initial state $\vec{w} \in \mathbb{F}^s$ in its internal memory gates and computes $(\vec{w}, \vec{y}) = \mathsf{prg}(\vec{w})$.



sub-circuit
Γ_2
Γ_2^0 (mini-circuit)
Γ_2^1 (mini-circuit)
Γ_2^2 (mini-circuit)
sub-circuit
Γ_1
Γ_1^0 (mini-circuit)
Γ_1^1 (mini-circuit)
Γ_1^2 (mini-circuit)
M (master)
sub-circuit
Γ_3
Γ_3^0 (mini-circuit)
Γ_3^1 (mini-circuit)
Γ_3^2 (mini-circuit)
test outputs
sub-circuit
$\Gamma_?$
$\Gamma_?^0$ (mini-circuit)
$\Gamma_?^1$ (mini-circuit)
$\Gamma_?^2$ (mini-circuit)

Figure 2: Transformed circuit (global view).

[4]In principle, for our application a passively secure 2-party protocol (e.g., [14], Chapter I, Section 4) would suffice. However, the security would need to rely on computational assumptions for the OT protocols, which would result in a less efficient scheme. In the following, the OT protocol is therefore performed by a third party, which samples an "OT-tuple", i.e., correlated randomness that is later used by the two other parties to perform secure computation.

[5]Notice that common implementations of PRGs do not out-

Evaluating the gates g of Γ by $(\Gamma^0, \Gamma^1, \Gamma^2)$

1. *Transformation for field addition, i.e., $\vec{a}\widehat{\oplus}\vec{b} = \vec{c}$:* Γ^1 holds the shares (a_1, b_1) that either were received from M via an in command, or resulted as an output from a previous gate. Similarly, Γ^2 holds (a_2, b_2). Given these inputs Γ^1 computes $c_1 = a_1 \oplus b_1$ and Γ_2 computes $c_2 = a_2 \oplus b_2$.
2. *Transformation for multiplication, i.e., $\vec{a}\widehat{\odot}\vec{b} = \vec{c}$:* This involves the mini-circuits $(\Gamma^0, \Gamma^1, \Gamma^2)$ and the driver circuit to forward commands between the circuits Γ^i. To keep the description simple, we will not explicitly describe the computation carried of by M as it only forwards commands. Initially, Γ^1 holds (a_1, b_1) and Γ^2 has (a_2, b_2). They proceed as follows:
   (a) Run jointly $(u, v) \leftarrow \mathsf{MultShares}(a_1, b_2)$ and $(u', v') \leftarrow \mathsf{MultShares}(b_1, a_2)$ (see description below).
   (b) *Mini-circuit Γ^1:* Compute $c_1 = a_1 \odot b_1 \oplus u \oplus u'$ and output c_1.
   (c) *Mini-circuit Γ^2:* Compute $c_2 = a_2 \odot b_2 \oplus v \oplus v'$ and output c_2.

Sub-circuit $(u, v) \leftarrow \mathsf{MultShares}(x, y)$

Initially, Γ^1 holds x and Γ^2 holds y. At the end Γ^1 holds u and Γ^2 has v such that $v = x \odot y \oplus u$ and $u \in \mathbb{F}$ defined below.

1. *Mini-circuit $\Gamma^0[\vec{w}]$:* Γ^0 has memory cells to store the internal state of the PRG $\vec{w}$. Notice that Γ^0 uses the contents of its memory cells $\vec{w}$ and computes $(\vec{w}, (u_1, u_2, u_3, u_4)) = \mathsf{prg}(\vec{w})$, where the output $\vec{w}$ represents the secret output. It then computes $u = u_3 \oplus u_4 \ominus u_1 \odot u_2$ and sends $((\mathsf{send}, 1), (u, u_2, u_3))$ and $((\mathsf{send}, 2), (u_1, u_4))$ to M.
2. *Mini-circuit Γ^2:* on receiving $((\mathsf{send}, 2), (u_1, u_4))$ from M, compute $z = y \oplus u_1$ and send $((\mathsf{send}, 1), z)$ to M.
3. *Mini-circuit Γ^1:* on receiving $((\mathsf{send}, 1), (u, u_2, u_3, z))$ from M, compute $e = (z \odot x) \oplus u_3$ and $f = x \oplus u_2$. Send $((\mathsf{send}, 2), (e, f))$ to M.
4. *Mini-circuit Γ^2:* on receiving $((\mathsf{send}, 2), (e, f))$ from M, compute $v = u_4 \oplus e \ominus f \odot u_1$.

Figure 3: The computation of the gates by the sub-circuits $(\Gamma_0, \Gamma_1, \Gamma_2)$. All operations are field operations in the underlying field $\mathbb{F}$. The $\mathsf{MultShares}$ circuit is used as sub-circuit in the field multiplication operation, where the latter is also shown in Figure 6 explaining the communication in further detail.

Here, $\vec{w}$ is the internal state of the PRG and $\vec{y}$ is the output. For our concrete construction, we require $\kappa := s + 4$. Notice that for security it does not matter how prg is implemented. Hence we misuse notation and let prg denote the circuit computing the PRG. Finally notice that to simplify the description all operations described in Figure 3 have fan-out 1. An extension to larger fan-out is trivially possible by just fanning out this single output.

Finally, in the *output post-processing phase*, we have that for each $i \in [\lambda]$ the sub-circuit Γ_i^1 sends $(\mathsf{out}, \vec{c}_i)$ and Γ_i^2 sends $(\mathsf{out}, \vec{d}_i)$ to M. Here, $(\vec{c}_i, \vec{d}_i)$ are λ independent sharings of the output $\vec{y}$ of Γ.

On receiving the out commands M proceeds as follows:

1. For each $i \in [\lambda]$ compute $\vec{y}_i = \vec{c}_i + \vec{d}_i$.
2. Output $\mathsf{MAJ}(\vec{y}_1, \ldots, \vec{y}_\lambda)$, where MAJ returns the most common value that occurs as an input; if two or more inputs are most common, then it outputs the first one of them. Notice that MAJ can easily be implemented using only standard arithmetic gates.

We additionally need to describe how to handle the initial secret state $\vec{m}$. The initialization function Init produces for each sub-circuit $i \in [\lambda]$, a secret sharing of $\vec{m}$ as $\vec{o}_i \leftarrow \mathbb{F}^s$ and $\vec{p}_i = \vec{m} - \vec{o}_i$, and stores $\vec{o}_i$ in the internal memory cells of Γ_i^1 and $\vec{p}_i$ in the internal memory cells of Γ_i^2, respectively. Notice that this implies that in total we require $2\lambda s$ memory cells in the transformed specification (compared to s in the original circuit Γ). Of course, the memory cells may be updated by the circuits (Γ_i^1, Γ_i^2) during the runs of the circuit. In the following description, we will often neglect mentioning the initial state explicitly as essentially it can be treated in the security analysis as part of the public inputs (this makes the adversary only stronger).

put random field elements, however, it is easy to do such a mapping in practice and we believe that the most common application of our techniques are binary circuits anyway, in which case we may just use AES in counter mode.

4.2 Correctness

Correctness of our construction follows by observing that the output of a transformed operation satisfies the invariant that it is a sharing of the corresponding value on the wire in Γ. The only non-trivial operation is the transformation $\widehat{\odot}$ of the field multiplication, which requires interaction between the mini-circuits. Hence, it results in connecting wires between the different Γ^j. We show that the transformation for the multiplication gate achieves correctness.

LEMMA 2. *For any $\vec{a}, \vec{b} \in \mathbb{F}^2$ we have $c_1 \oplus c_2 = (a_1 \oplus a_2) \odot (b_1 \oplus b_2)$, where $(c_1, c_2) = \vec{a}\widehat{\odot}\vec{b}$ is the output of the transformed multiplication operation.*

The proof of this lemma appears in the extended version of this paper [15]. To complete the correctness analysis, observe that each of the sub-circuits Γ_i produces a sharing of an output $\vec{y}_i$. When M receives the out command it will re-combine the two shares to recover $\vec{y}_i$ and compute $\mathsf{MAJ}(\vec{y}_1, \ldots, \vec{y}_\lambda)$. Due to the correctness of the computation phase all of them will be identical, and $\mathsf{MAJ}(\vec{y}_1, \ldots, \vec{y}_\lambda)$ outputs the correct result $\vec{y} \leftarrow \Gamma(\vec{x})$. It is straightforward to extend the correctness analysis to circuits that have secret inputs/outputs. We can just view the secret inputs/outputs as an additional public input/output of the circuit.

4.3 Testing circuits

Besides the circuit transformation that outputs a protected specification that supposedly is implemented by the malicious manufacturer, the trojan protection scheme also defines a tester T. The description of T is public, and uses a probabilistic approach to defeat the malicious manufacturer $\mathcal{A}$. Consider the (potential) malicious implementation $\{\mathsf{D}_i\}_i \leftarrow \mathcal{A}(1^k, (\mathsf{M}, \{\Gamma_i\}_i))$ output by $\mathcal{A}$. Following Figure 2, our construction consists of λ sub-devices, each of them made of three mini-devices $(\mathsf{D}_i^0, \mathsf{D}_i^1, \mathsf{D}_i^2)$ which supposedly implement the mini-circuits Γ_i^j. As the sub-devices D_i operate independently, we can test them independently.

Let $\mathsf{D}_i = (\mathsf{D}_i^0, \mathsf{D}_i^1, \mathsf{D}_i^2)$ be one of the sub-devices. Denote the joint view of the mini-devices D_i^j by $\mathsf{View}(\mathsf{D}_i(\vec{r}, \vec{s}))$ when run as part of D_i on public inputs $(\vec{r}, \vec{s})$ after the initialization with $\vec{m}$. Notice that in this view we have all tuples of the form $(\mathsf{cmd}, \mathsf{val})$ exchanged between the mini-devices D_j^i and the master circuit M. As the outputs of D_i are also sent as a command, the view also contains the output shares $(\vec{c}_i, \vec{d}_i)$. Similarly, we denote by $\mathsf{View}(\Gamma_i(\vec{r}, \vec{s}))$ the view of the mini-circuits Γ_i^j when run on public inputs $(\vec{r}, \vec{s})$.

At a high-level, T repeats the following process for each $i \in [\lambda]$. First, it chooses a random value $t_i \leftarrow [t]$, where t_i denotes the number of test runs. In each of the t_i runs the public/secret inputs are chosen uniformly at random and we execute once D_i produced by $\mathcal{A}$ and once the specification Γ_i (in both cases using the same inputs). If the views differ in one of the runs we return false and the tester T aborts. The formal description of the tester T is given in Figure 4.

```
The tester $\mathsf{T}^{\mathsf{D}_1(\cdot),\ldots,\mathsf{D}_\lambda(\cdot)}(1^k, \lambda)$

Set the initial state $\mathsf{Init}(\{\mathsf{D}_i\}_i, \vec{0})$
For $i \in [\lambda]$ repeat the following:
    Sample $t_i \leftarrow [t]$ and repeat for $t_i$ times:
        Sample random sharing of public input $\vec{r}, \vec{s} \leftarrow \mathbb{F}^\alpha$
        If $\mathsf{View}(\mathsf{D}_i(\vec{r}, \vec{s})) \neq \mathsf{View}(\Gamma_i(\vec{r}, \vec{s}))$ return false
Return true.
```

Figure 4: The tester T for verifying whether the devices follow the specification given by Γ'.

4.4 Main theorem and security proof

The basic idea for the security of our construction is as follows. Recall that in the testing phase each of the sub-devices D_i was tested for a random number of times $t_i \in [t]$. Consider a mental experiment where instead of running the sub-devices for t_i times, we execute them for $t_i + n$ times, i.e., we view the real runs of D_i also as test runs. Informally, the malicious manufacturer $\mathcal{A}$ wins in the mental experiment if the sub-devices D_i succeed in the test runs, but he makes a large fraction of D_i fail in the following n real runs. We show that the probability that $\mathcal{A}$ wins in the mental experiment decreases exponentially with the number of sub-devices.

What remains to show is that for the devices D_i the real environment, where $\mathcal{A}$ can choose the inputs, looks (computationally) indistinguishable from the test runs. In particular, we need to avoid that the adversary can choose the inputs $\vec{x}_i$ in order to signal to the devices that they are now used outside of the test environment.[6] The basic idea to prevent such signaling is to let the mini-devices D_i^j run a passively secure 3-party computation protocol on shares of the input. This guarantees that none of the mini-devices actually knows the inputs on which it computes, and can start to behave differently from the test environment. The rest of this section is structured as follows. In Section 4.4.1 we prove that the specification of our construction satisfies the property that real runs and test runs are indistinguishable. In Section 4.4.2 we use this fact to prove robustness.

[6]Consider the input trigger attack where the adversary chooses a 128-bit random value at production time on which the device is starting to deviate when received as input.

```
The experiment $\mathsf{Real}_\Gamma^j(1^k, q, \{\vec{x}_i\}_{i\in[q]}, \vec{m})$:
Initialize circuit $\Gamma$ by $\mathsf{Init}(\Gamma, \vec{m})$
Output $\big(\mathsf{View}(\Gamma^j(\mathsf{Share}(\vec{x}_1))), \ldots, \mathsf{View}(\Gamma^j(\mathsf{Share}(\vec{x}_q)))\big)$

The experiment $\mathsf{Random}_\Gamma^j(1^k, q)$:
Initialize circuit $\Gamma$ by $\mathsf{Init}(\Gamma, \vec{0})$
Sample $\vec{z}_1, \ldots, \vec{z}_q \leftarrow \mathbb{F}^\alpha$ uniformly at random
Output $\big(\mathsf{View}(\Gamma^j(\mathsf{Share}(\vec{z}_1))), \ldots, \mathsf{View}(\Gamma^j(\mathsf{Share}(\vec{z}_q)))\big)$
```

Figure 5: Views produced by a continuous real and test execution of the specification Γ^j by the mini-circuits of our construction.

4.4.1 *The transformed specification*

Before we move to the device-level, we prove a property about the transformed specification $\Gamma_i := (\Gamma_i^0, \Gamma_i^1, \Gamma_i^2)$. Recall that each Γ_i is independent from the other sub-circuits and specifies exactly the same functionality. Hence, we concentrate in the following on a single sub-circuit and omit to explicitly mention the parameter i. Let Γ denote one of the sub-circuits and Γ^j are the corresponding mini-circuits.

In Figure 5 we define two distributions: The distribution $\mathsf{Real}_\Gamma^j(1^k, q, \{\vec{x}_i\}_{i\in[q]}, \vec{m})$ considers the view of Γ^j on public inputs $\vec{x}_i$ and with secret initial input $\vec{m}$. On the other hand $\mathsf{Random}_\Gamma^j(1^k, q)$ describes the view of Γ^j in q runs of Γ with random inputs. The next lemma states that both distributions are computationally close.

LEMMA 3. *Let $q \in \mathbb{N}$ denote the number of executions. For any $j \in [3]$, any set of public inputs $\{\vec{x}_i\}_{i\in[q]}$, and any initial secret input $\vec{m} \in \mathbb{F}^k$, we have:*

$$\mathsf{Real}_\Gamma^j(1^k, q, \{\vec{x}_i\}_{i\in[q]}, \vec{m}) \approx_c \mathsf{Random}_\Gamma^j(1^k, q).$$

The proof appears in the extended version of this paper [15].

4.4.2 *Trojan robustness of our construction*

The theorem below shows the robustness of our construction. In particular, it states that trojan robustness increases exponential with the number of devices.

THEOREM 1. *Let $t, n, \ell, k \in \mathbb{N}_{>0}$ with $n < t$ and k being the computational security parameter.* $\Pi = (\mathsf{TR}, \mathsf{T})$ *is (t, n, ϵ)-trojan robust for $\epsilon := \left(\frac{n}{t}\right)^{\lambda/2} + \mathsf{negl}(k)$.*

We provide the proof in the extended version of this paper [15]. Here we briefly discuss the parameters given by the theorem statement. The factor $\mathsf{negl}(k)$ can be ignored since it comes from the security of the PRG. The dominating factor for realistic values of $t, n, \ell := 3\lambda$ is the value $\left(\frac{n}{t}\right)^{\lambda/2}$. Let us give an example for the level of robustness we can achieve. Suppose we have $\lambda = 10$ sub-circuits, which results into 30 mini-devices that need to be produced by the manufacturer. Suppose we test each of the sub-devices for max $t = 10^9$ runs (which is realistic for simple hardware devices), and want to use them for $n = 10^5$ executions later. The theorem guarantees that except with probability 10^{-20} the resulting computation is correct.

4.5 Stateful and randomized circuits

So far we only discussed how to handle original circuits Γ that are stateless (i.e., write their internal state only once) and are deterministic (i.e., have no rand gates). We now briefly discuss how to extend our results to probabilistic and stateful circuits. To handle the rand gates we do a simple transformation before using our compiler TR. Namely, we replace each rand gate by the output of a deterministic PRG. Clearly, this reduces probabilistic computation to the deterministic case we already discussed in the previous sections. However, if the original circuit Γ was stateless, then after replacing the rand gates in Γ by the PRG, the new circuit Γ' may become stateful. Hence, to complete our construction we need to discuss how to handle stateful primitives (e.g., like a PRG or a stream cipher).

The problem with the stateful primitives is that in the testing phase we test the different sub-devices a different number of times, which makes the devices end up (after the testing) in different states. When then after the testing phase in the real-execution we execute all sub-devices jointly (driven by the master) and take the majority of their outputs, then the majority will fail to provide the correct result (since all the sub-devices are in a different state). A first idea to deal with this issue is to reset the state of the sub-devices after the testing. However, such reseting can be noticed by the malicious sub-devices.[7]

To overcome this problem, we augment Γ with a special circuit that carries out input-triggered re-initialization. For instance, consider a Boolean circuit Γ that originally takes inputs from $\{0,1\}^\alpha$. We add an additional bit to the input which signals re-initialization, i.e., the inputs of the augmented Γ are now from $\{0,1\}^{\alpha+1}$, where if the first bit is 1, then the following α bits are used to reset the state. Otherwise, if the first bit is 0, then the it is a normal execution of Γ without updating the initial state. Such augmented circuitry can easily be implemented using our simple operations from above. For instance, using the simple example above, we compute for re-initialization: $\vec{m} = (1-b) \odot \vec{m} \oplus b \odot \vec{x}$, where b is the first bit of the input, $\vec{m}$ is the initial state and $\vec{x}$ are remaining α bits of the input. Clearly, if $b = 0$ the state is not touched, while if $b = 1$ we rewrite the state with the input. In order to achieve security, we then compile Γ added with the augmented circuitry described above using our compiler TR. Notice that this means not that we also share the triggering bit b as otherwise the sub-devices can notice that they are re-initialized.

Of course, the above approach has one drawback. It gives the adversary in the real-execution the possibility to overwrite the state with a fresh adversarial chosen state. If $\vec{m}$ is a key then this is an undesirable feature. However, this can be easily fixed by telling the master M to set the first bit permanently to 0 after re-initialization.

[7]For instance, the devices may just monitor their internal state in some extra memory and hence can notice if the state was changed outside of their normal execution pattern.

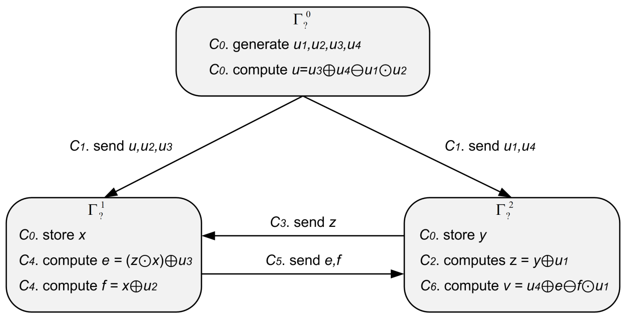

$\Gamma_?^0$
C_0. generate u_1,u_2,u_3,u_4
C_0. compute $u=u_3\oplus u_4\ominus u_1\odot u_2$
C_1. send u,u_2,u_3
C_1. send u_1,u_4
$\Gamma_?^1$
C_0. store x
C_4. compute $e = (z\odot x)\oplus u_3$
C_4. compute $f = x\oplus u_2$
C_3. send z
C_5. send e,f
$\Gamma_?^2$
C_0. store y
C_2. computes $z = y\oplus u_1$
C_6. compute $v = u_4\oplus e\ominus f\odot u_1$

Figure 6: MultShares with threee mini-circuits.

5. DISCUSSIONS

In this section, we discuss the relevance of our circuit model, the implementation cost of our transformed circuits and testing phase, and the concrete attacks covered by our threat model. Due to place constraints, we focus on general observations and arguments in favor of the practicality of our proposals and leave the concrete investigation of meaningful case studies as a scope for further research.

5.1 Instantiation of the circuit model

In practice, the circuit specification of Section 2.1 can be simply instantiated with existing Hardware Description Languages (HDLs) such as VHDL or Verilog, and its communication commands with standard communication interfaces. In fact, the only fundamental requirement for this circuit specification is that it allows describing and testing the functional correctness of the devices implementing them.

Besides, since for our previous construction, we essentially convert the original circuit Γ into a couple of passively secure 3-party implementations of this circuit, we use an abstract representation based on addition and multiplication gates, which allow us to describe a generic compiler. Yet, this is not a strict requirement and any specialized compiler that would lead to a more efficient 3-party implementation of a given circuit Γ (as long as it can be specified in a hardware description language) is in fact eligible.

5.2 Cost of the transformed circuits

Concretely, our circuit transformation essentially requires to design λ sub-circuits, each of them corresponding to a 3-party implementation of the functionality to protect. For linear functionalities (in the binary/arithmetic field we consider) this implies overheads that are linear in the total number of devices ℓ. So as usual in multiparty computation, the most significant overheads come from the non-linear operations. In order to estimate these overheads, an implementation of the $\mathsf{MultShares}$ circuit of Figure 3 is sketched in Figure 6, where we can see that such an operation can be carried out in 6 "abstract cycles" (denoted from C_0 to C_6 on the figure) with a PRG and 10 arithmetic operations.

Therefore, in terms of timing/latency the best that we can hope is a cycle count that is proportional to the logic depth of the functionality to protect, which would happen if we compute all the multiplications in parallel. Considering that all the communications have to commute through the master circuit, and that each send, in,out command can be performed in c cycles, the latency of each multiplicative level will be multiplied by a maximum factor $6c$ (since not all the abstract cycles require communications).

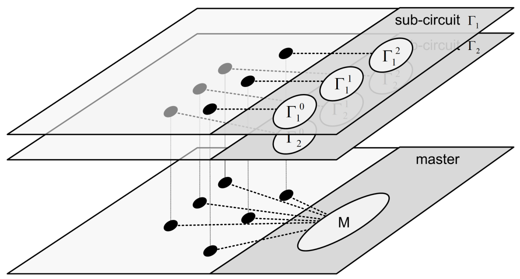


Figure 7: Implementation with 3D circuits.

In terms of circuit size, each sub-circuit will require a (constant) multiplicative overhead ($\approx \times 10$) due to the arithmetic operations of MultShares, and a (constant) additive overhead due to the PRG. The impact of the latter naturally depends on the implementation size of this PRG compared to the one of the functionality to protect. Taking the (expensive) case where we compute several multiplications in parallel, we could for example require to generate 128 pseudorandom bits per cycle with an AES-based PRG, which remains achievable, e.g., in low-cost FPGA devices.

Quite naturally, there may be additional overheads due to representation issues. For example, standard block ciphers are generally implemented thanks to table lookups, which are not included in our circuit model. In this respect, we first note that such overheads can be mitigated by taking advantage of cryptographic primitives designed for masking, multiparty computation or fully homomorphic encryption (which aim to minimize the multiplicative complexity and depth of the circuits) [18, 4]. Besides, even for a standard cipher such as the AES, the broad literature on masking suggests that 3-party implementations similar to ours are achievable in mainstream embedded devices (see, e.g. [25, 17] for software and hardware evaluations).

Eventually, we show in the next section that much more efficient specialized solutions can be obtained for certain important cryptographic functionalities.

5.3 Testing of the transformed circuits

As clear from the previous section, the security of our trojan-resilient circuits depends on the possibility to test sub- and mini-circuits, including all their communications. In general, this can be implemented by connecting various circuits to a master via standard communication interfaces. However, we note that more compact solutions also exist, by taking advantage of the 3D technologies of which the usefulness for trojan-resilient circuits was already put forward in [20]. As illustrated in Figure 7, we can then easily embed the sub-circuits as the different tiers of a 3D hardware.

Besides, note that (as suggested in the right part of Figure 2), one can speed up the communication between the mini-circuits by allowing them to communicate directly, given that the tester can monitor these communications with "wires" that would be used only during the testing phase, and of which the monitoring would not be noticed by the mini-circuits, i.e., under a "no hidden communications" assumption. This could be achieved by equiping the tester with specialized hardware capacities (e.g., an oscilloscope).

5.4 Attacks & limitations

We conclude this section by listing the attacks covered by our threat model and its limitations.

Compared to [30], we prevent any digital input-triggered hardware trojan (e.g., single-shot cheat codes and sequence cheat codes). In this respect, we additionally cover the risk of "infection attacks", where one activated sub-circuit starts to communicate with others sub-circuits, which is achieved by limiting the communication between them.

Next, we prevent internally-triggered trojans (e.g., time bombs) in a more general manner than [30]. Namely, this previous work was limited to preventing volatile time bombs with power resets. We also prevent non-volatile ones (e.g., a counter that would store the number of executions of the circuit independent of its powering) thanks to our testing phase. We believe this is an important improvement for emerging technologies such as FRAM-based devices [16].

We also cover all the attacks considered in [20] and, as previously mentioned, are able to efficiently bound the success rate of these attack to exponentially small probabilities.

By contrast, as mentioned in Section 2.2, we cannot prevent physical trojan attacks since our testing phase is looking for functional incorrectness. Yet, we note that exploiting physical side-channels such as the power consumption or electromagnetic radiation of a chip usually requires physical proximity (which may be excluded by other means). As for side-channels that are exploitable remotely, such as timing attacks [11], they could be prevented by functional testing (e.g., in order to ensure constant-time executions). In general, the extension of our tools towards physical hardware trojans is an important scope for further research.

Eventually, we mention one more type of attack which, to the best of our knowledge, has not been mentioned in the literature so far and is not covered by our tools, namely "battery attacks". In this case, the infected chip would go on performing harmful operations (e.g., the increaing of a counter) independent of whether the chip is performing any computation. Interestingly, existing (e.g., lithium) battery and energy harvesting technologies are currently based on quite different design techniques than digital ASICs [12, 27]. So it may be a reasonable hardware assumption to ask such trojans to be detected by chip inspection (via microscopy or other means), which we leave as another interesting challenge for hardware research.

6. EFFICIENT FUNCTIONALITIES

In this section, we briefly discuss how to use testing amplification to get better efficiency for certain cryptographic primitives. We achieve the better efficiency by (a) focusing on specific functionalities and (b) by only showing a weaker security property. In particular, in contrast to trojan robustness from Definition 1, which aims at correctness, we will focus on a security property that is tailored to the particular functionality we want to protect. Notice that typically the constructions presented in this section do not achieve correctness and do not protect against the denial-of-service attacks mentioned in the introduction. That is, a hardware trojan can always disable the functionality completely.

6.1 Trojan secure PRGs

We first describe how to construct a PRG that is *trojan secure*, where "trojan security" is a weaker security guarantee than trojan robustness from Definition 1. Nevertheless, we argue that for certain cryptographic primitives and certain applications trojan security is a sufficiently strong security property. In contrast to trojan robustness which requires essentially that the malicious devices output correct results (i.e., the same result as the honest specification), trojan security of a PRG only guarantees that the malicious implementation of the PRG still outputs pseudorandomness.

Constructing a trojan secure PRG is very simple. Just let the malicious manufacturer produce ℓ device $\mathsf{D}_1, \ldots, \mathsf{D}_\ell$, where each D_i supposedly implements a cryptographically strong PRG with binary output $\{0,1\}^\beta$.[8] Each of the D_i's is initialized with a random and independent initial secret seed K_i. The master M then runs the devices D_i and just XORs the outputs of D_i on each invocation. Observe that since all keys K_i were sampled uniformly and independently and we XOR the outputs of D_i, we get that the output of the composed device is pseudorandom as long as one device D_i outputs pseudorandomness.

Let us now argue about the security of the above construction. Testing the above implementation is easy: we just use the same random testing approach as for our circuit compiler. That is, each of the sub-devices D_i is tested independently for t_i times. Next, we can use a similar analysis as in Theorem 1 to show that if the D_i's pass the testing phase, then with probability $1 - (n/t)^\ell$ at least one device outputs the correct result for all n real executions. By the above observation, this suffices to show that with probability at least $1 - (n/t)^\ell$ the device outputs pseudorandomness.[9]

De-randomization of our circuit compiler. In our circuit compiler, the master M is randomized since it needs to secret-share the inputs of the device (which requires randomness). We can use the above construction of the trojan secure PRG to de-randomize M. To this end we let the malicious manufacturer produce ℓ additional devices, where each computes a PRG. Whenever M needs uniform randomness, we replace it by the output of the above construction of a trojan secure PRG. Notice that this further simplifies the assumptions that we put on M, since now the master M does not need to run a trusted component for random number generation. In this approach the complexity of M is reduced to a small number of additions and multiplications.

6.2 Other cryptographic primitives

We conclude our paper with a short discussion on other cryptographic primitives that can benefit from the technique of testing amplification (i.e., having many independent devices that are tested independently and the combined using a master). For efficiency, we concentrate on the "trojan security" (see Sect. 6.1 above) and because of the space reasons, we only discuss how to construct an efficient trojan secure Message Authentication Code (MAC).

Recall that a message authentication code is a symmetric cryptographic primitive that can be used to guarantee the authenticity of messages. One way to protect a MAC against trojan attacks is to use our generic compiler from Section 4. We now describe a more efficient way achieving trojan security for MACs. Let us start by describing the security property we are aiming at. Let D be a device that supposedly implements a secure MAC with key K, i.e., it outputs tags with respect to the key K. Informally, trojan security guarantees that valid tags can only be produced by running the device D. Notice that this in particular implies that an adversary interacting with the supposedly malicious D in the n real executions does not learn anything about the internal secret key K. More concretely, to specify the trojan security of a MAC, we consider the following two phases (of course, prior to these two phases we execute a testing phase of the sub-devices):

1. In the *learning phase*, $\mathcal{A}$ interacts with the potentially malicious implementation D. That is, $\mathcal{A}$ can ask for MACs of messages of his choice and sees the output of the MAC. Notice that this can be done for at most n times (similar as in the robustness definition).
2. In the *challenge* phase the adversary has to provide a forgery for the key K and a fresh message X.

In order to construct an efficient trojan secure MAC, we proceed as follows. Let $F : \{0,1\}^k \times \{0,1\}^\alpha \to \{0,1\}^\beta$ be a secure pseudorandom function (for instance, instantiated with an AES). We let the malicious manufacturer produce ℓ sub-devices $\mathsf{D}_1, \ldots, \mathsf{D}_\ell$ where each supposedly implements the PRF F. The sub-devices D_i are then combined by the master M in the following way. On an input message $X \in \{0,1\}^\alpha$ the master produces an ℓ-out-of-ℓ secret sharing $(X_1, \ldots, X_\ell)$ of X. Each share X_i is given to the sub-device D_i as input, which computes $Y_i = F(K_i, X_i)$. The value Y_i is given back to the master M who computes $Y = \bigoplus_i Y_i$ and outputs the tag $((X_1, \ldots X_\ell), Y)$. Notice that we can de-randomize the master M by using our PRG construction from Section 6.1. Verification of the tag produced by the above construction is simple. Essentially, since $(X_1, \ldots X_\ell)$ are part of the tag the verifier can use $(K_1, \ldots, K_\ell)$ to verify the correctness of the MAC. The above construction has the shortcoming that it increases the length of the tag by ℓ times the message length. We leave it as an interesting open question to improve the tag length.

The basic intuition why the above construction is trojan secure is as follows. First, observe that the sub-devices D_i operate independently from each other (they all use independent keys and no communication is needed between the D_i's for computing F). Second, they are run on shares of the inputs X, so the adversary cannot initiate malicious behavior by signaling it through the inputs. The random testing guarantees that with probability $1 - (n/t)^\ell$ at least one device D_i outputs the correct result for all n real executions. Since we are XORing the outputs of all sub-devices D_i, we are guaranteed that as long as at least one device D_i operates honestly, it "blinds" the outputs of all other devices, and hence hides the output of potential malicious devices (that try to reveal their internal keys).

In general it can be observed that, informally speaking, in order to construct efficient trojan robust cryptographic primitives using our technique of testing amplification, we need algorithms that are *both* input homomorphic and key homomorphic (essentially this is what the use of the MPC enables). We leave it as an interesting question for future work to find such cryptographic schemes.

Acknowledgments. Stefan Dziembowski is supported by the Foundation for Polish Science. Sebastian Faust is funded by the Emmy Noether Program FA 1320/1-1 of the German

[8]It also may be a elements in a field, but we only consider the most simple case here.

[9]Observe that we obtain better parameters than for the strong property of trojan robustness since we only require that one sub-device behaves honestly. This allows us to save a factor of 1/2 in the exponent.

Research Foundation (DFG). François-Xavier Standaert Standaert is a research associate of the Belgian Fund for Scientific Research (FNRS-F.R.S.).

References

[1] J. Aarestad, D. Acharyya, R. M. Rad, and J. Plusquellic. "Detecting Trojans Through Leakage Current Analysis Using Multiple Supply Pad I_{DDQ} s". In: *IEEE Trans. Information Forensics and Security* 4 (2010).

[2] S. O. Adee. "The Hunt For The Kill Switch". In: *IEEE Spectrum* 5 (May 2008). ISSN: 0018-9235.

[3] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. "Trojan Detection using IC Fingerprinting". In: *IEEE S&P*. 2007.

[4] M. R. Albrecht, C. Rechberger, T. Schneider, T. Tiessen, and M. Zohner. "Ciphers for MPC and FHE". In: *EUROCRYPT*. 2015.

[5] G. Ateniese, A. Kiayias, B. Magri, Y. Tselekounis, and D. Venturi. *Secure Outsourcing of Circuit Manufacturing.* Cryptology ePrint Archive, Report 2016/527. 2016.

[6] C. Bayer and J.-P. Seifert. "Trojan-resilient circuits". In: *PROOFS*. 2013.

[7] S. Bhunia, M. S. Hsiao, M. Banga, and S. Narasimhan. "Hardware Trojan attacks: threat analysis and countermeasures". In: *Proceedings of the IEEE* 8 (2014).

[8] E. Biham, Y. Carmeli, and A. Shamir. "Bug Attacks". In: *CRYPTO*. 2008.

[9] E. Biham and A. Shamir. "Differential Fault Analysis of Secret Key Cryptosystems". In: *CRYPTO*. 1997.

[10] D. Boneh, R. A. DeMillo, and R. J. Lipton. "On the Importance of Eliminating Errors in Cryptographic Computations". In: *J. Cryptology* 2 (2001).

[11] B. B. Brumley and N. Tuveri. "Remote Timing Attacks Are Still Practical". In: *ESORICS*. 2011.

[12] C. K. Chan, H. Peng, G. Liu, K. McIlwrath, X. F. Zhang, R. A. Huggins, and Y. Cui. "High-performance lithium battery anodes using silicon nanowires". In: *Nature nanotechnology* 1 (2008).

[13] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *CRYPTO*. 1999.

[14] R. Cramer. "Introduction to Secure Computation". In: *Lectures on Data Security, Modern Cryptology in Theory and Practice, Summer School, Aarhus, Denmark, July 1998*. 1998.

[15] S. Dziembowski, S. Faust, and F.-X. Standaert. *Private Circuits III: Hardware Trojan-Resilience via Testing Amplification.* Cryptology ePrint Archive. 2016.

[16] G. Fox, F Chu, and T Davenport. "Current and future ferroelectric nonvolatile memory technology". In: *Journal of Vacuum Science & Technology B* 5 (2001).

[17] V. Grosso, F. Standaert, and S. Faust. "Masking vs. multiparty computation: how large is the gap for AES?" In: *J. Cryptographic Engineering* 1 (2014).

[18] V. Grosso, G. Leurent, F. Standaert, and K. Varici. "LS-Designs: Bitslice Encryption for Efficient Masked Software Implementations". In: *FSE*. 2014.

[19] S. K. Haider, C. Jin, M. Ahmad, D. M. Shila, O. Khan, and M. van Dijk. *Advancing the State-of-the-Art in Hardware Trojans Detection.* Cryptology ePrint Archive, Report 2014/943. 2014.

[20] F. Imeson, A. Emtenan, S. Garg, and M. V. Tripunitara. "Securing Computer Hardware Using 3D Integrated Circuit (IC) Technology and Split Manufacturing for Obfuscation". In: *USENIX Security Symposium*. 2013.

[21] Y. Ishai, A. Sahai, and D. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *CRYPTO*. 2003.

[22] Y. Ishai, M. Prabhakaran, A. Sahai, and D. Wagner. "Private Circuits II: Keeping Secrets in Tamperable Circuits". In: *EUROCRYPT*. 2006.

[23] P. C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO*. 1996.

[24] P. C. Kocher, J. Jaffe, and B. Jun. "Differential Power Analysis". In: *CRYPTO*. 1999.

[25] A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. "Pushing the Limits: A Very Compact and a Threshold Implementation of AES". In: *EUROCRYPT 2011*. 2011.

[26] S. Narasimhan, D. Du, R. S. Chakraborty, S. Paul, F. G. Wolff, C. A. Papachristou, K. Roy, and S. Bhunia. "Hardware Trojan Detection by Multiple-Parameter Side-Channel Analysis". In: *IEEE Trans. Computers* 11 (2013).

[27] S. Priya and D. J. Inman. *Energy harvesting technologies.* 2009.

[28] M. Tehranipoor and F. Koushanfar. "A Survey of Hardware Trojan Taxonomy and Detection". In: *IEEE Design & Test of Computers* 1 (2010).

[29] R. S. Wahby, M. Howald, S. Garg, abhi shelat, and M. Walfish. *Verifiable ASICs.* Cryptology ePrint Archive, Report 2015/1243. 2015.

[30] A. Waksman and S. Sethumadhavan. "Silencing Hardware Backdoors". In: *IEEE S&P*. 2011.