zk-Sherlock: Exposing Hardware Trojans in Zero-Knowledge

Dimitris Mouris
ECE, University of Delaware
jimouris@udel.edu

Charles Gouert
ECE, University of Delaware
cgouert@udel.edu

Nektarios Georgios Tsoutsos
ECE, University of Delaware
tsoutsos@udel.edu

***Abstract*—As integrated circuit (IC) design and manufacturing have become highly globalized, hardware security risks become more prominent as malicious parties can exploit multiple stages of the supply chain for profit. Two potential targets in this chain are third-party intellectual property (3PIP) vendors and their customers. Untrusted parties can insert hardware Trojans into 3PIP circuit designs that can both alter device functionalities when triggered or create a side channel to leak sensitive information such as cryptographic keys. To mitigate this risk, the absence of Trojans in 3PIP designs should be verified before integration, imposing a major challenge for vendors who have to argue their IPs are safe to use, while also maintaining the privacy of their designs before ownership is transferred. To achieve this goal, in this work we employ modern cryptographic protocols for *zero-knowledge proofs* and enable 3PIP vendors prove an IP design is free of Trojan triggers without disclosing the corresponding netlist. Our approach uses a specialized circuit compiler that transforms arbitrary netlists into a zero-knowledge-friendly format, and introduces a versatile Trojan detection module that maintains the privacy of the actual netlist.**

***Index Terms*—Circuit simulation, Hardware security, Hardware Trojans, IP verification, Zero-knowledge proofs (ZKPs)**

I. Introduction

Integrated Circuit (IC) designs are embedded in most electronic equipment, and as a result, IC security has become of crucial importance as the globalized economy heavily relies on System-on-Chip (SoC) designs. The IC supply chain depends on procuring a variety of Intellectual Property (IP) cores from third-party vendors (3PIP) and integrating them with components that are designed in-house to fabricate the IC [1]. However, the integrity of these externally developed IPs cannot always be guaranteed, and as a result malicious actors can potentially inject hardware Trojans to the SoC designs. Such malicious modifications in hardware could be triggered under certain conditions (e.g., user input, time-based, etc.) or be always on [2]. Rarely-activated Trojans are typically programmed to engage only under a unique set of circumstances created by an attacker and are hard to detect when in their dormant state [3]. When activated, Trojans can alter device functionality by influencing output wires or creating a side channel through which sensitive data can be leaked. It is critical for the IC supply chain to address security concerns, such as hardware Trojans, instead of solely on functionality and runtime performance.

IP core verification is a crucial step of SoC design [4], during which IP consumers provide functional requirements to the vendors, and the 3PIP vendors design circuits that meet these specifications.[1] The goal of IP core verification is to convince system integrators about the functionality of the generated 3PIP designs. Thus, ensuring that the circuit is compliant with the specified constraints while achieving a high degree of testability is a crucial consideration in the IC supply chain. Most common solutions include formal logic verification [5], simulation-based methods [6], and application-specific instruction-set processors [7]. Previous solutions are mostly geared toward IP *functional* verification, but fail to protect the privacy of the IP designs. However, recent efforts have also focused on using cryptographic protocols such as homomorphic encryption and zero-knowledge proofs (ZKP) to enhance the security of transactions in the IC supply chain. [8], [9] securely outsource the evaluation of 3PIP netlists to third parties to ensure the confidentiality of the circuit inputs, however, the actual netlist is still visible since homomorphic encryption does not provide functional privacy (only data privacy). [10], [11] preserve the privacy of the netlist using ZKPs but only focus on functional verification and fail to address the increasingly important issue of hardware Trojans. Indeed, existing solutions offer no support for system integrators to confirm that the IP they are purchasing does not contain any malicious modifications, without inspecting it themselves.

There are two classes of defenses against hardware Trojans that both require access of the IP, namely invasive and non-invasive [1]. The former incurs significant costs as it requires expensive equipment and renders the IP unusable afterwards; the latter relies on functional and statistical IC testing, such as path-delay measurements [12], and gate-level characterization [13]. Such defenses require unrestricted access to the IP, as well as the statistical distribution of gate characteristics.

In this work, we propose zk-Sherlock, a novel framework for detecting hardware Trojans in zero-knowledge (ZK), i.e., *without allowing access to the circuit*. Our methodology introduces a custom ZK-friendly algorithm for Trojan detection that resolves the deadlock between 3PIP vendors and IP customers. This deadlock is created by the mutual distrust between vendors and consumers: vendors may withhold an IP before receiving payment from customers to avoid the risk of IP theft, while customers may refuse to purchase an IP until they are convinced that it satisfies their requirements. A key

[1] We use the terms IP consumer and system integrator interchangeably.

contribution of zk-Sherlock is the translation of a netlist into a ZK-friendly format that enables testing using public input vectors to detect any *gates with the least switching activity* and argue about the presence of potentially malicious logic. Our main observation is that the majority of logic gates in a netlist would switch for most input pairs. zk-Sherlock leverages this observation to tally the total number of switched gates across multiple evaluations with different inputs and detect the ones that have not switched. We evaluate our approach using multiple benchmarks from ISCAS '85 and '89 [14], [15] with judiciously injected Trojans following the methodology of [16] (as in the TRIT benchmarks on Trust-Hub [17]).

II. Preliminaries

A. Background on Zero-Knowledge Proofs

A zero-knowledge proof (ZKP) is a cryptographic protocol between a *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$ in which the former can prove to the latter that a statement is true without revealing any additional information about that statement. Specifically, the *zero-knowledge* property guarantees that if the statement is true, a cheating $\mathcal{V}$ does not learn anything after interacting with an honest $\mathcal{P}$, other than the fact that the statement is true. If $\mathcal{V}$ is honest and the statement is true, the ZKP must be *complete*, so that $\mathcal{V}$ will always be convinced about the statement by an honest $\mathcal{P}$. Finally, a ZKP is *sound* if a cheating $\mathcal{P}$ cannot convince an honest $\mathcal{V}$ to accept a statement, except with negligible probability [18]. ZKPs enable multiple real-world applications, from private crowdsourcing to private blockchain transactions and anonymous auctions [19]–[21].

More formally, let us assume a public state machine $\mathbb{SM}$ executing a procedure with a public input $\mathbb{x}$ and secret input $\mathbb{w}$ (only known to $\mathcal{P}$), which return a public output $\mathbb{y}$ such that $\mathbb{y} = \mathbb{SM}(\mathbb{x}, \mathbb{w})$; here $\mathcal{V}$ knows everything but $\mathbb{w}$. In a ZKP, $\mathcal{P}$ runs the state machine $\mathbb{SM}$ locally while hiding the value of $\mathbb{w}$ and provides cryptographic guarantees (i.e., the proof) to $\mathcal{V}$ that all the state transitions were performed correctly. $\mathcal{V}$ can then verify the proof and become convinced that $\mathcal{P}$ knows a correct *witness* $\mathbb{w}$. Some ZKP constructions rely on *arithmetic circuits*, such as [22], while others rely on the *random-access machine (RAM)* model (e.g., [23]). RAM protocols record the *execution trace* of the machine as a sequence of intermediate states, each comprising multiple blocks of registers. In the generated trace, all state transitions satisfy special cryptographic constraints that are mathematically expressed as low-degree polynomials over a finite field and should hold during the entire execution of $\mathbb{SM}$. Other constructions that use arithmetic circuits require a trusted pre-processing phase that binds $\mathcal{P}$ and $\mathcal{V}$ to a static arithmetic circuit for each different $\mathbb{SM}$. A significant advantage of the RAM-based protocols such as "Scalable Transparent ARgument of Knowledge" [20], [24] is that they are universal, i.e., they do not depend on a specific state machine and can verify any state transition.

B. Threat Model

In this work, we assume threats originating at any point in the IC supply chain, from the design phase to IP integration.

Cheating $\mathcal{P}$: A cheating $\mathcal{P}$ (i.e., the IP vendor) has financial incentives to deceive the IP consumer by falsely claiming that they possess an IP with certain functional specifications while the IP could be embedded with malicious circuitry at certain locations. In one scenario, $\mathcal{P}$ may try to deceive an honest $\mathcal{V}$ by trying to sell an IP with a Trojan that alters the agreed-upon functionality. In another scenario, $\mathcal{P}$ has performed malicious gate modifications to the IP while still meeting the agreed-upon functionality. The system integrator (i.e., buyer) has to test the IP with multiple input vectors, and also verify the correctness of the ZKP. $\mathcal{P}$ succeeds if they can break the soundness of the protocol with probability greater than $2^{-\lambda}$ (where λ is zk-Sherlock's security parameter) by producing a fake ZKP that will convince an honest $\mathcal{V}$ to accept it.

Cheating $\mathcal{V}$: We assume a cheating $\mathcal{V}$ (i.e., an IP consumer) that follows the protocol but also have incentives to extract information about the private IP from an honest $\mathcal{P}$. More specifically, $\mathcal{V}$ may attempt to extract and learn the IP netlist before paying, even though the vendor wants to keep that netlist secret until payment is received. In other words, $\mathcal{V}$ wants to break the *zero-knowledge* property of the protocol and learn the private IP $\mathbb{w}$, which is infeasible if $\mathcal{P}$ follows the protocol faithfully. Notably, $\mathcal{V}$ can only learn that $\mathbb{SM}(\mathbb{x}, \mathbb{w}) = \mathbb{y}$ (in our case $\mathbb{SM}$ checks if the IP $\mathbb{w}$ is Trojan-free).

III. Zero-Knowledge Trojan Detection

A. Overview of our Methodology

In this work, we present zk-Sherlock, a novel methodology for privacy-preserving hardware Trojan detection. Our approach enables 3PIP vendors to prove to system integrators that: a) they possess an IP with some predefined functional specifications, and b) the IP is Trojan-free without revealing anything about its netlist. zk-Sherlock utilizes zero-knowledge proofs to detect any hardware Trojan triggers by inspecting the switching activity of every gate and identify any non-switching logic *without sharing the netlist* with the IP consumer. To that end, we have designed a specialized ZK state machine $\mathbb{SM}$ that consumes a netlist as private input $\mathbb{w}$ and proves that the circuit has not been embedded with malicious logic. More specifically, $\mathbb{SM}$ is a public *circuit simulator* that evaluates gates and records their switching activity for different input/output pairs. Most gates in a netlist will flip after relatively few input sets, unless these gates trigger a rarely-activated Trojan. Thus, our observation is that after evaluating a small number of possible input pairs with zk-Sherlock, all gates should have switched (with high probability), except for Trojan logic. zk-Sherlock then offers provable guarantees on the computational integrity of $\mathbb{SM}$, i.e., that the circuit simulation was performed faithfully. A high-level overview of zk-Sherlock is depicted in Fig. 1 and discussed in the following paragraphs.

1) Privacy-Preserving Functional Verification: To solve the mutual distrust between 3PIP vendors ($\mathcal{P}$) and IP consumers ($\mathcal{V}$), zk-Sherlock first needs to prove that a secret IP adheres to some predetermined specifications. As shown in Fig. 1(a), the untrusted 3PIP vendor synthesizes an IP described in a

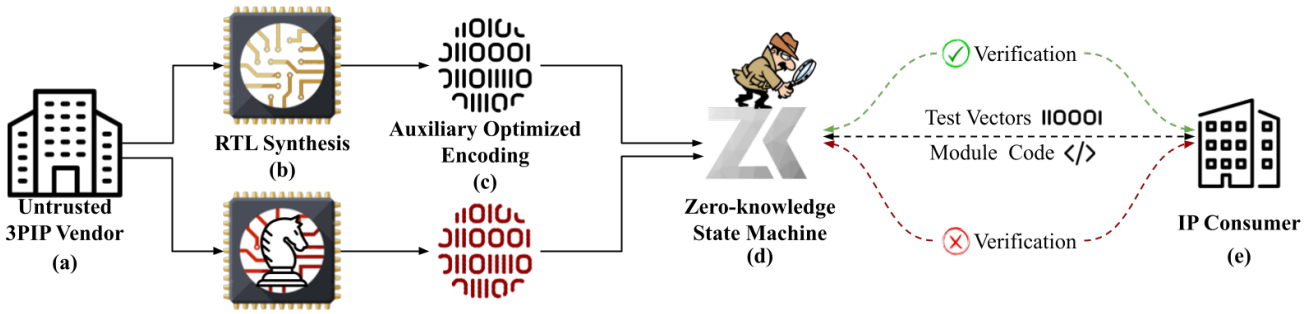


Fig. 1. **Overview of zk-Sherlock:** (a) $\mathcal{P}$ possesses an IP described in a Hardware Description Language that has some agreed-upon functional specifications. (b) The 3PIP vendor ($\mathcal{P}$) synthesizes the IP and generates a gate-level netlist, determining the correct evaluation order of the gates. (c) The 3PIP vendor transforms the IP into a ZK-friendly encoding for the Trojan detection state-machine $\mathbb{SM}$. (d) $\mathcal{P}$ executes $\mathbb{SM}$ using the netlist as private input and public test vectors chosen by $\mathcal{V}$. (e) The two parties interact and $\mathcal{P}$ convinces $\mathcal{V}$ that the IP is Trojan-free and that $\mathbb{SM}$ was evaluated correctly.

Hardware Description Language (HDL),[2] creates a gate-level netlist, and then uses the specialized compiler of zk-Sherlock to transform it into a ZK-friendly encoding, as shown in Fig. 1(c). More specifically, our compiler guarantees that this encoding can be evaluated sequentially (i.e., one gate at a time) and does not have any inter-dependencies between the gate inputs and the outputs from previous gates, thus it can be evaluated by zk-Sherlock's ZK state machine. During the third step (in Fig. 1(c)), we apply various optimization techniques to minimize the total number of intermediate wires by employing graph-coloring techniques. Next, $\mathcal{P}$ and $\mathcal{V}$ agree on the functional simulation algorithm $\mathbb{SM}$, shown in (d), and the latter provides public input vectors $\mathbb{x}$. Finally, the 3PIP vendor runs $\mathbb{SM}$ locally to simulate the private netlist $\mathbb{w}$ with inputs $\mathbb{x}$ and computes a public circuit output $\mathbb{y}$, which is then checked by $\mathcal{V}$ along with the cryptographic proof that every step of $\mathbb{SM}$ was executed faithfully.

2) Privacy-Preserving Trojan Detection: After proving that the IP adheres to some agreed-upon functional specifications, zk-Sherlock needs to prove that the netlist does not contain any Trojan trigger logic. As malicious parties aim to leak sensitive information or induce errors by inserting Trojans in circuits that are triggered under special conditions, the gates that comprise the injected Trojan logic are *nearly-unused* and are *rarely activated* on common inputs. Towards that end, we have expanded the state machine $\mathbb{SM}$ to also analyze the gate-switching activity (i.e., when the output signal of the gate flips) of all the gates in the circuit across multiple inputs in order to detect malicious triggers. An important contribution of our work is that $\mathbb{SM}$ acts both as a circuit simulator (to test the netlist functionality in ZK), and as gate switching activity analyzer at the same time, combining functional verification and Trojan detection. The circuit compiler of zk-Sherlock translates netlist gates into a sequence of $\mathbb{SM}$ instructions that track which gates have low switching activity over different input pairs, and flags them as potentially malicious. Contrary to existing hardware Trojan detection mechanisms, our ZKP approach hides the IP netlist from the verifier.

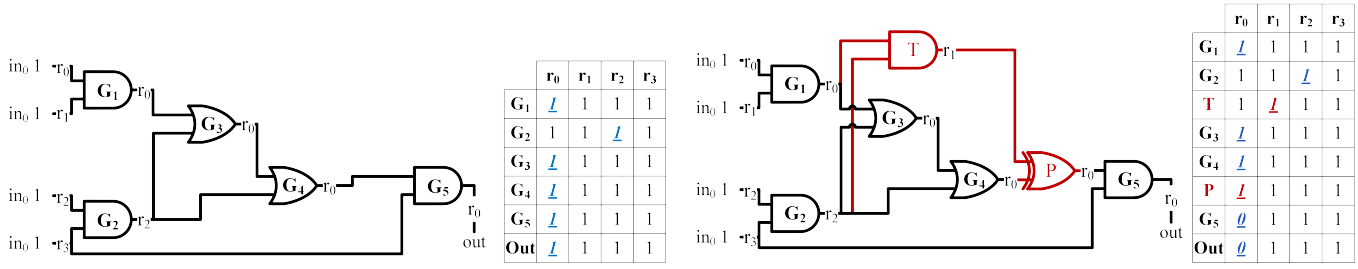
Fig. 2 demonstrates how a hardware Trojan can affect the functionality of a circuit under only a single input combination. To clarify how this encoding is evaluated, we show a two-dimensional table on the right-hand side of both (a) and (b) that represent four $\mathbb{SM}$ registers ($r_0 - r_3$) and how the values in these registers change after the evaluation of each gate. For example, all registers are initialized with "1" and after the evaluation of G_1, its output "1" is written to r_0 (underlined in the table). In Fig. 2 (b), we observe that the trigger T of the Trojan is only activated when all inputs are "1", which only happens in one out of 2^4 different input combinations, rendering gate T the gate with the rarest switching activity. However, we do not observe a similar behaviour for the payload gate P, which switches for various inputs (e.g., 1100). The intuition of zk-Sherlock is motivated from the aforementioned observation, i.e., determine the gates with a suspiciously low switching activity as they can potentially be part of a Trojan trigger.

3) zk-Sherlock back-end: zk-Sherlock utilizes the Zilch framework [20] as the cryptography back-end to argue about the correctness of $\mathbb{SM}$. Internally, zk-Sherlock leverages the MIPS-like assembly programming language of Zilch to implement the state machine, which simulates the evaluation of a circuit and computes all gate switching activity. At a technical level, Zilch enforces different cryptographic constraints that should hold during each $\mathbb{SM}$ transition and are used to prove the correctness of the execution of $\mathbb{SM}$ in zero-knowledge.

The initial state of the zk-Sherlock state machine is filled with multiple blocks, each with 64 1-bit registers initialized to zero. With every gate evaluation, the $\mathbb{SM}$ modifies at most one register and copies all the blocks into a new state. The sequence of states forms a two-dimensional table that represents the *execution trace* of the $\mathbb{SM}$ (Fig. 2). The type and the index of the update on the $\mathbb{SM}$ state correspond to the different operation that is performed in ZK. For instance, an arithmetic operation (e.g., addition) will modify the state in a different way than a bitwise operation (e.g., `AND`, `OR`).

Our state machine consumes *private* inputs that correspond to the netlist description (known only to $\mathcal{P}$) and *public* test vector inputs that $\mathcal{V}$ provides. Utilizing the Zilch back-end, zk-Sherlock cryptographically asserts that all state machine transitions were performed in accordance to the operation in the encoded netlist $\mathbb{w}$; this is enforced using polynomial constraints over the transitions that assert their satisfiability to

[2]Without loss of generality, zk-Sherlock uses Verilog.



	r_0	r_1	r_2	r_3
G_1	*1*	1	1	1
G_2	1	1	*1*	1
G_3	*1*	1	1	1
G_4	*1*	1	1	1
G_5	*1*	1	1	1
Out	*1*	1	1	1

(a) Trojan-free circuit.

	r_0	r_1	r_2	r_3
G_1	*1*	1	1	1
G_2	1	1	*1*	1
T	1	*1*	1	1
G_3	*1*	1	1	1
G_4	*1*	1	1	1
P	*1*	1	1	1
G_5	*0*	1	1	1
Out	*0*	1	1	1

(b) Circuit embedded with a Trojan.

Fig. 2. The gates are labeled by the evaluation order (first G_1, then G_2, etc.) and are also shown on the rows of the tables (execution trace). The underlined values in the tables show which simulation variable was overwritten after the evaluation of the gate. The variables $r0 - r3$ represent four $\mathbb{SM}$ registers, also shown at the outputs of the gates. (a) shows a circuit that outputs "1" when all four inputs are set to high (*note:* there exist more combinations to output "1"). (b) shows the same circuit as (a) after being injected with an example Trojan that is only activated when all inputs are set to "1".

$\mathcal{V}$. Effectively, zk-Sherlock convinces the IP consumer that a secret netlist has certain functional specifications and that it is Trojan-free without revealing its composition.

B. Serialized Encoding for State Machine

To generate a proof, zk-Sherlock synthesizes HDL programs into netlists consisting of Boolean gates and flip-flops. Our approach utilizes the Yosys Open SYnthesis Suite for RTL synthesis to generate Electronic Design Interchange Format (EDIF) netlists based on Verilog files [25]. We optimize the netlist during RTL synthesis with Yosys by converting the entire circuit into standard two-input logic gates and removing unused wires. Next, zk-Sherlock associates each logic gate with specific input and output wires (as shown in Fig. 2) with $\mathbb{SM}$ registers (e.g., $r_0 - r_3$). An important step for our compiler is to identify gate dependencies by creating a directed acyclic graph, running a topological sort to eliminate dependencies, and finally assigning unique $\mathbb{SM}$ registers to the wires. For instance, in order to evaluate the gate G_3 in Fig. 2(a), gates G_1 and G_2 have to be evaluated first. Notably, our compiler applies an additional optimization to further reduce the total number of registers used, as the state size (i.e., the total number of registers in $\mathbb{SM}$) can impact the execution time of our ZK back-end. Finally, the zk-Sherlock compiler serializes the circuit as a sequence of MIPS-like instructions for Zilch, which is ultimately passed as the private input $\mathbb{w}$ to $\mathbb{SM}$, which evaluates the serialized netlist for a given test vector.

The approach depicted above is directly applicable to combinational circuits, but sequential circuits need additional considerations. Since sequential circuits require more than one clock cycle to evaluate completely, we need multiple iterations over a given netlist. Thus, we unroll the circuit for the desired number of clock cycles and when a flip flop is encountered during evaluation, we propagate its input signal to its output signal at the next clock cycle. While this approach correctly emulates sequential circuits, it is suboptimal when used with our zk-Sherlock module. Often, certain combinational logic segments do not change between clock cycles and do not need to be re-evaluated every time. Therefore, we omit gates that are not connected to an upstream flip-flop to account for this and avoid redundant computations.

Without loss of generality, Fig. 3 shows the serialized encoding for an IP with 64 gates (we only show the first, second, and last gate for simplicity) and how zk-Sherlock tracks the gate switching activity (shown as "Switching Gates") across different iterations. Our encoding consists of: gate type (e.g., `AND`), unique gate identifier (depends on number of gates in the IP), register block ID#, and offset inside the register block for the two input wires and the output wire. The state encoding is organized into multiple blocks (each block holds 64 bits), and each bit represents one $\mathbb{SM}$ register. For instance, the first line of the encoding reads the 10^{th} and the 12^{th} bits of the first block as inputs and writes at the 13^{th} bit of the first 64-bit block. For simplicity, in Fig. 3 we only assume two register blocks: the "Switching Gates" block tracks the gate identifiers that have switched, while the "Gate Outputs" block stores the gate outputs. At the end of the first cycle simulation, some gates have switched and this is reflected in the bottom left block, along with gate outputs in this cycle (bottom right). As more cycles are evaluated, the "Switching Gates" blocks will continue to reflect the number of gates that have encountered a state change. After a specified number of iterations, the presence of a zero in this block suggests a potential Trojan trigger. The total number of iterations is decided by $\mathcal{V}$, so that more iterations offer increased assurance against Trojans.

C. State Machine Evaluation

1) Output generation: We simulate the circuit as follows: First, zk-Sherlock reads a Boolean gate from $\mathbb{w}$ and performs the corresponding Boolean operation on the two input wires

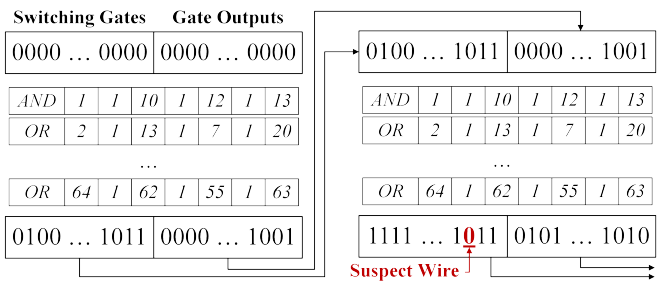


Fig. 3. Abstraction of two cycles of an 64-gate circuit in zk-Sherlock. "Switching Gates" block and "Gate Outputs" block keep track of the switched gates and the gate outputs, respectively.

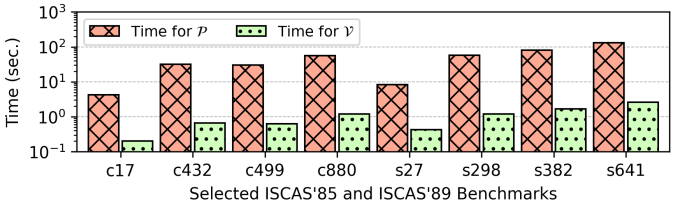


Fig. 4. Experimental timings for $\mathcal{P}$ and $\mathcal{V}$ per input-pair for selected benchmarks.

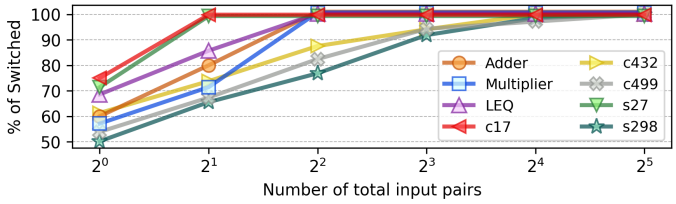


Fig. 5. Percentage of the total gates that switched over increasing number of input pairs for selected Trojan-free benchmarks.

to compute the correct output. All gates with no dependencies (i.e., first layer gates) get inputs directly from the public input $\mathbb{x}$, while all intermediate and output gates read their inputs from the outputs of preceding gates which are encoded in different registers in the $\mathbb{SM}$. Our compiler guarantees that the input and output $\mathbb{SM}$ registers map to the correct gates. This way, $\mathbb{SM}$ can keep track of the identifiers of the specific gates that have switched. Internally, we use multiple register blocks (depicted in Fig. 3 as "Switching Gates"), where each register corresponds to the switching activity of a particular gate.

2) Computational Integrity: To prevent a malicious $\mathcal{P}$ from deceiving an honest $\mathcal{V}$ by switching the IP under test across different executions, zk-Sherlock applies a secure hash function to compute two secure digests: one for the IP netlist itself and one for the state registers. Then, at the beginning of each new cycle, $\mathcal{P}$ initializes the $\mathbb{SM}$ registers with the previous execution state and computes the secure hash of the new registers to prove that she propagated the state from the previous cycle. $\mathcal{P}$ evaluates the circuit again, keeping track which gates switched during the current and all previous executions. At the same time, $\mathcal{P}$ recomputes the hash of the private IP and proves to $\mathcal{V}$ that the same $\mathbb{w}$ was used across all the executions. Notably, $\mathcal{V}$ does not learn any intermediate results about the gate switching activity as the intermediate registers are stored at the beginning of $\mathbb{w}$.

IV. Experimental Results

We evaluate our methodology using selected benchmarks from the ISCAS'85 and ISCAS'89 suites [14], [15]. During synthesis, we used the `proc`, `flatten`, `synth`, and `abc -g simple` flags in Yosys to generate circuits with standard 2-input logic gates. Moreover, our zk-Sherlock compiler is implemented in Python 3, while the state machine simulator is implemented in a MIPS-like assembly language for ZKPs. As the back-end of zk-Sherlock can benefit from multiple threads to accelerate the proving time, we used an m5.24xlarge AWS EC2 instance featuring two Intel Xeon Platinum 8175M processors at 2.5 GHz.

In a realistic scenario, zk-Sherlock uses multiple input test-vectors chosen by $\mathcal{V}$, which correspond to the functional properties the IP consumer wants to assert. The input vector values do not impact the proving time, but affect how the gates switch. Because $\mathcal{V}$ has no knowledge of the underlying circuit, the best method for choosing input vectors is to generate them randomly. As shown in our experiments, random test vectors cause all the gates in a circuit to flip relatively quickly with high probability when a Trojan is not present.

Fig. 4 shows the amortized execution time for both the prover and verifier for a set of inputs. In the case of sequential circuits (from the ISCAS'89), the presented time reflects the cost per cycle. In practice, multiple input vectors should be utilized at the discretion of $\mathcal{V}$ to minimize the risk of false positives (where one or more non-malicious gates have yet to switch across all input sets provided). The timings for $\mathcal{P}$ scale linearly with the number of logic gates per circuit and the execution time for $\mathcal{V}$ scales logarithmically with increasing circuit sizes. In addition, the size of the execution trace scales with the number of $\mathbb{SM}$ registers required to simulate the circuit, which can impact the execution time. Generally, the wider the circuit, the more register blocks are required; for instance, c17 requires 5 intermediate wires at its widest point, which can fit into a single 64-bit packed block, while c880 requires 87 registers that fit into two 64-bit blocks. Notably, the proving times for all circuits are independent of the input pattern, as an identical set of gates is visited for each run.

The verifier needs to choose enough test vectors to avoid false positives. While each circuit's gate switching activity will vary, in Fig. 5 we investigate the percentage of possible input pairs required to flip all non-malicious gates in an assortment of eight circuits with random input wire sets. We found that 32 input sets were sufficient to flip all gates, and hence result in no false positives, in all eight Trojan-free circuits, while all but one circuit had flipped earlier. However, random inputs assume that $\mathcal{V}$ has no knowledge about the *behavior* of the circuit, whereas practically, the prospective buyer of an IP is aware of the expected functionality. Thus, $\mathcal{V}$ can cleverly choose pairs to cover a large number of cases in a small number of input pairs; for instance, in a simple example with a 2-bit less-than-or-equal circuit, $\mathcal{V}$ can choose two sets of inputs that will cause all of the gates to switch. If a Trojan is hidden in such a circuit, our approach can detect it with just 2 input pairs.

To further assess our approach, we injected Trojans in c17, c432, c499, s27, and s298 ISCAS circuits following the methodology of TRIT [16].[3] We carefully modified the benchmarks so that the Trojans get triggered with only a rare combination of the input wires and alter specific output wires, similarly to Fig. 2. zk-Sherlock successfully detected the Trojans in all cases; interestingly, we note that c499, the largest combinational circuit tested, yielded the lowest probability of false negatives for a given number of inputs, as there are 41 input wires and hence 2^{41} possible inputs and only 32 inputs were required to flip all benign wires. Therefore, since the

[3]Precompiled TRIT benchmarks from Trust-Hub [17] are incompatible with Yosys so we created equivalent benchmarks based on their ISCAS circuits.

Trojan can only trigger on a very rare input combination, the chance of the Trojan flipping during the tested input sets is approximately zero. For smaller circuits, such as c17 and s27, there are far fewer possible input pairs (32 and 128 respectively), so the probability of false negative increases to approximately 6% for c17 and 1.5% for s27 as each require 2 input pairs to successfully flip all benign wires.

V. Related Work

zk-Sherlock focuses on hardware Trojans that are activated based on particular user inputs and thus the trigger nodes are rarely executed. The authors of [26] use an advised genetic algorithm to generate test vectors to detect Trojans based on rare nodes. Similarly, FANCI [27] performs functional analysis to flag logic that is unlikely to affect the circuit outputs, whereas VeriTrust [28] identifies potential Trojan wires by examining verification corner cases. All of these techniques assume that the circuit design is available for inspection and that details such as the statistical distribution of gate characteristics are known. A critical benefit of zk-Sherlock is that it proves to the IP consumer that a netlist is Trojan-free without revealing anything about it.

Pythia [10], [11] describes an approach related to zk-Sherlock by introducing the problem of functional IP verification in zero-knowledge. However, these works focus on ensuring that the IP has some functional properties and that it also satisfies constraints related to area, performance, and power consumption by checking different input-output pairs provided by IP consumers. Orthogonal to these earlier works, our approach transforms the problem of IP verification to a zero-knowledge protocol, and introduces a powerful encoding and a versatile $\mathbb{SM}$ that allows tracking the switching activity to offer assurance about hardware Trojan triggers.

VI. Concluding Remarks

In this paper, we have proposed a unique methodology for detecting hardware Trojans in zero-knowledge (i.e., without having access to the IP netlist). zk-Sherlock proposes a new encoding that enables evaluating both the circuit and computing the gate switching activity at the same time. Using this gate activity, our methodology identifies the nodes that are triggered under rare conditions, and thus flags them as potentially malicious logic. In effect, zk-Sherlock enables 3PIP vendors to convince system integrators that a netlist is Trojan-free, so that integrator have only black-box access to the IP by submitting test vectors. Additionally, as the system integrator may provide inputs, zk-Sherlock employs a secure hashing to confirm that across different executions: (a) the same secret netlist was used, and (b) the gate-switching activity counters were propagated correctly. Our experiments with ISCAS'85 and '89 benchmarks demonstrate that our approach converges quickly to "Suspected Trojan" or "Trojan-free" classification for an IP under test.

Acknowledgment

This work was supported by the University of Delaware Research Foundation under Grant 21A01012.

References

[1] M. Rostami *et al.*, "Hardware security: Threat models and metrics," in *ICCAD*. IEEE/ACM, 2013, pp. 819–823.
[2] R. Karri *et al.*, "Trustworthy hardware: Identifying and classifying hardware Trojans," *IEEE Computer*, vol. 43, no. 10, pp. 39–46, 2010.
[3] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.
[4] A. Deshpande, "Verification of IP-Core based SoC's," in *ISQED*. IEEE, 2008, pp. 433–436.
[5] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *VTS*. IEEE, 2012, pp. 252–257.
[6] G. Moretti *et al.*, "Your Core – My Problem? Integration and Verification of IP," in *DAC*. IEEE/ACM, 2001, pp. 170–171.
[7] M. Stadler *et al.*, "Functional verification of intellectual properties (IP): a simulation-based solution for an application-specific instruction-set processor," in *IEEE ITC*, 1999, pp. 414–420.
[8] C. Konstantinou, A. Keliris, and M. Maniatakos, "Privacy-preserving functional IP verification utilizing fully homomorphic encryption," in *DATE*. EDAA, 2015, pp. 333–338.
[9] C. Gouert and N. G. Tsoutsos, "ROMEO: Conversion and Evaluation of HDL Designs in the Encrypted Domain," in *DAC*. ACM/EDAC/IEEE, 2020, pp. 1–6.
[10] D. Mouris and N. G. Tsoutsos, "Pythia: Intellectual Property Verification in Zero-Knowledge," in *DAC*. ACM/EDAC/IEEE, 2020, pp. 1–6.
[11] D. Mouris, C. Gouert, and N. G. Tsoutsos, "Privacy-preserving IP Verification," *IEEE TCAD*, 2021.
[12] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2008, pp. 51–57.
[13] Y. Alkabani and F. Koushanfar, "Consistency-based characterization for ic trojan detection," in *2009 ICCAD*, 2009, pp. 123–127.
[14] F. Brglez, "A neural netlist of 10 combinational benchmark circuits," *IEEE ISCAS: Special Session on ATPG and Fault Simulation*, pp. 151–158, 1985.
[15] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS*. IEEE, 1989, pp. 1929–1934.
[16] J. Cruz *et al.*, "An automated configurable trojan insertion framework for dynamic trust benchmarks," in *DATE*, 2018, pp. 1598–1603.
[17] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *ICCD*. IEEE, 2013, pp. 471–474.
[18] M. Bellare and O. Goldreich, "On Defining Proofs of Knowledge," in *CRYPTO*. Springer, 1992, pp. 390–420.
[19] A. Kosba *et al.*, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Symp. on Security and Privacy (SP)*. IEEE, 2016, pp. 839–858.
[20] D. Mouris and N. G. Tsoutsos, "Zilch: A framework for deploying transparent zero-knowledge proofs," *IEEE TIFS*, vol. 16, pp. 3269–3284, 2021.
[21] ——, "Masquerade: Verifiable Multi-Party Aggregation with Secure Multiplicative Commitments," Cryptology ePrint Archive, Report 2021/1370, 2021.
[22] B. Parno *et al.*, "Pinocchio: Nearly practical verifiable computation," in *Symp. on Security and Privacy (SP)*. IEEE, 2013, pp. 238–252.
[23] E. Ben-Sasson *et al.*, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *CRYPTO*. Springer, 2013, pp. 90–108.
[24] ——, "Scalable zero knowledge with no trusted setup," in *CRYPTO*. Springer, 2019, pp. 701–732.
[25] C. Wolf, J. Glaser, and J. Kepler, "Yosys - A Free Verilog Synthesis Suite," in *Austrochip*, 2013.
[26] M. Nourian, M. Fazeli, and D. Hély, "Hardware trojan detection using an advised genetic algorithm based logic testing," *JETTA*, vol. 34, no. 4, pp. 461–470, 2018.
[27] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *CCS*. ACM, 2013, pp. 697–708.
[28] J. Zhang *et al.*, "VeriTrust: Verification for Hardware Trust," *IEEE TCAD*, vol. 34, no. 7, pp. 1148–1161, 2015.