Privacy-Preserving IP Verification

Dimitris Mouris, Charles Gouert and Nektarios Georgios Tsoutsos

Abstract

The rapid growth of the globalized integrated circuit (IC) supply chain has drawn the attention of numerous malicious actors that try to exploit it for profit. One of the most prominent targets of such parties is the third-party intellectual property (3PIP) vendors and their circuit designs. With the increasing number of transactions between vendors and system integrators, the threat of IP reuse and piracy has become a significant consideration for the IC industry. What is more, the correctness of 3PIP designs should be verified before integration, imposing another challenge for 3PIP vendors since they have to prove the functionality of their designs to system integrators while protecting the privacy of the circuit implementations. To eliminate this deadlock, we utilize the cryptographic technique of "zero-knowledge proofs" to enable 3PIP vendors to convince system integrators about various functional properties of a circuit (e.g., area, power, frequency) without disclosing its netlist (i.e., *in zero-knowledge*). Our approach comprises a circuit compiler that transforms arbitrary netlists into a zero knowledge-friendly format and a library of modules that provide cryptographic guarantees for various properties of the netlist while hiding the actual gates. We evaluate our method using combinational and sequential circuits from the ISCAS and ITC benchmark suites.

I. Introduction

The advent of the Internet of Things (IoT) has made possible an extensive set of applications such as transportation systems, healthcare, home automation, and many more [1], [2]. Because of the demand for small-size and low-power IoT devices, manufacturers have adopted more compact and energy-efficient hardware design paradigms. As a result, System-on-Chip (SoC) has conquered the market from multi-chip designs by combining lower power and area consumption with increased reliability and functionality, all into a single, integrated chip (IC). In the contemporary IC supply chain, some components are designed in-house and are integrated with a variety of Intellectual Property (IP) cores from third-party vendors in order to fabricate the IC [3], [4].

The authors are with the Department of Electrical and Computer Engineering, University of Delaware, Newark, DE. E-mail: {jimouris, cgouert, tsoutsos}@udel.edu

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

The ever-growing demand for IC-based solutions results in an increase of third-party IP (3PIP) vendors that try to maximize their profits by providing design standards and guidelines and also by making their IPs reusable so they can be utilized by multiple design layouts [5]. 3PIP cores such as digital signal processors (DSPs) and FFT engines are commonly adopted by chip designers to serve specific purposes in the overall system. This widespread appropriation of 3PIP becomes an attractive target for malicious users and rogue foundries that attempt to trick honest parties and steal their IPs for financial gain [6]. Attackers continue to find novel and elaborate ways to illegally pirate an IP, such as system-level analysis [7] and reverse engineering [8]. Thus, finding solutions to IP piracy and drawing the attention of the IC supply chain to security instead of solely functionality and performance is a major concern.

In the modern IC business model, an essential step of SoC design is IP core verification [9], [10]. First, system integrators (i.e., IP consumers) provide some functional requirements to the 3PIP vendors, who in turn design circuits that meet these specifications. The goal of IP core verification is to prove the functionality of the generated 3PIP designs to system integrators. However, confirming that the circuit complies with the specified properties while achieving high testability is a major challenge in the IC supply chain. Researchers have come up with various solutions, namely application-specific instruction-set processors [11], formal logic verification [12], satisfiability (SAT) solvers [13], and simulation-based methods [14]. Yet, all these solutions focus more on the functional verification of IP designs rather than protecting their privacy. Recent research directions leverage homomorphic encryption to securely outsource the evaluation of the 3PIP netlist to a third party [15]–[17]. However, these approaches only protect the input vectors: the netlist designs can still be visible since homomorphic operations, in this case, do not provide functional privacy (i.e., only data privacy).

Motivated by the lack of netlist-level privacy preservation, we propose a novel method that utilizes zero-knowledge (ZK) proofs to resolve the deadlock of mutual distrust between 3PIP vendors and IP consumers. In this case, the former withholds an IP until a financial agreement is made (due to the IP piracy risk), while the latter refuses to purchase an IP without being convinced that it meets all of the agreed-upon specifications. To instantiate our strategy, we developed the Pythia framework that enables system integrators to verify that a *potentially untrusted* 3PIP vendor possesses an IP that satisfies some agreed-upon properties *without having access to it* (i.e., gaining "zero" knowledge about the IP). One key contribution in Pythia is the conversion of a netlist into a zero knowledge-friendly format that can be evaluated with public test input vectors and generate a public output along with a proof which verifies that the output was computed faithfully.

The back-end of our framework relies on a state-of-the-art ZK protocol called "Scalable Transparent ARgument of Knowledge" (zk-STARK) [18], [19] to implement a library of special state machine modules that can evaluate both combinational and sequential circuits in zero-knowledge and further prove various functional properties such as performance, area, and power. Developing these modules as ZK state machines enables Pythia to argue about their computational integrity and offer *provable guarantees* that the IC netlist properties are verified faithfully. Pythia allows the evaluation of 3PIP netlists as Boolean circuits without revealing the netlist itself by encoding input test vectors into a judiciously selected format compatible with our ZK state machine modules. Each state machine consumes a 3PIP netlist as private (secret) input and test vectors as public inputs; Pythia executes the corresponding ZK state machine on these inputs and generates cryptographic proofs asserting that the output of evaluating the private netlist with the public inputs is correct and computed faithfully.

Overall, in this work we claim the following contributions:

- **Circuit compiler:** Development of a compiler that can automatically translate any Boolean circuit (combinational and sequential) into a serialized encoding that can be interpreted by our ZK state machines. Our novel compiler resolves any inter-dependencies between intermediate gate inputs and upstream gate outputs.
- **ZK state machine modules:** Design of a library of ZK state machines that can evaluate Boolean circuits on any input test vector while preserving the privacy of the netlist itself. On top of this functionality, the developed ZK modules can also prove in zero-knowledge various properties of the netlist, such as estimated area, critical path delay, and expected power consumption.
- **Optimizer:** Performance enhancements by utilizing bit-packing and graph-coloring techniques for optimal allocation of the intermediate wires of Pythia's ZK state machine vector. Furthermore, Pythia automatically divides the compiled 3PIP netlist into independent shares that can be evaluated *in parallel*, while cryptographically proving continuity between all these consecutive shares.

Roadmap: In Section II we discuss background notions such as zero-knowledge proofs, pseudorandom functions, and our threat model, while in Sections III-V we elaborate on our privacy-preserving IP verification methodology, including our special ZK modules and performance optimizations. We present the evaluation of our framework in Section VI, followed by related work in Section VII. Finally, our concluding remarks are summarized in Section VIII.

II. Cryptography Background

A. Zero-Knowledge Proofs Primer

Zero-knowledge proofs (ZKPs) are cryptographic primitives involving two parties: a (potentially untrusted) *prover* $\mathcal{P}$ and a *verifier* $\mathcal{V}$. With ZKPs, $\mathcal{P}$ can prove to $\mathcal{V}$ knowledge of a secret (known as the *witness*) that satisfies certain properties [20]. ZKPs enable a plethora of applications such as privacy-preserving computation, electronic voting, blockchains, and others [21], [22]. In a simple, yet powerful example, $\mathcal{P}$ wants to prove that she knows a preimage (secret witness $\mathbb{w}$) for a hash digest chosen by $\mathcal{V}$, without revealing what the preimage is [23]. To formalize the above statement, let us assume an algorithm $\mathbb{A}$ that implements a cryptographic hash function (e.g., SHA-256), compares the computed output to a public input $\mathbb{x}$ (i.e., the hash digest chosen by $\mathcal{V}$) and outputs a Boolean value $\mathbb{y}$ whether the two hashes match or not:

$$\mathbb{y} = \mathbb{A}(\mathbb{x}, \mathbb{w}) = \begin{cases} True, & \text{if SHA-256}(\mathbb{w}) \equiv \mathbb{x} \\ False, & \text{otherwise} \end{cases}$$

In a zero-knowledge protocol, $\mathcal{P}$ computes the value $\mathbb{A}(\mathbb{x}, \mathbb{w})$ locally without revealing $\mathbb{w}$, and provides strong cryptographic guarantees to $\mathcal{V}$ that $\mathbb{A}$ was evaluated faithfully. If the result is $True$, then $\mathcal{V}$ is convinced that $\mathcal{P}$ knows a correct witness $\mathbb{w}$ to make SHA-256 return $\mathbb{x}$. The most prominent ZKP constructions rely either on *arithmetic circuits* [24], [25] or on *random access machines (RAM)* [26]–[29]. The former class requires an expensive, trusted pre-processing phase that binds the two parties to a static arithmetic circuit for each different $\mathbb{A}$. Conversely, RAM-based protocols are more powerful and flexible as they do not depend on a specific computation and can verify any state transition in the RAM.

RAM-based ZK protocols: In a typical RAM-based protocol, the prover $\mathcal{P}$ evaluates locally a publicly-known algorithm $\mathbb{A}$ using the public $\mathbb{x}$ and private $\mathbb{w}$ input values, and records the *execution transcript* of the RAM as a sequence of intermediate states of the state machine. In the generated transcript, all state transitions satisfy special cryptographic constraints that are expressed as low-degree polynomials over a finite field $\mathbb{F}$ and should hold during the execution of $\mathbb{A}(\mathbb{x}, \mathbb{w})$.

By design, *zero-knowledge* protocols ensure that nothing is revealed about $\mathbb{w}$ beyond the fact that $\mathcal{V}$ has faithfully executed $\mathbb{A}(\mathbb{x}, \mathbb{w})$ and the output is $\mathbb{y}$. Additionally, ZKP systems satisfy two important properties: *completeness* and *soundness*. Completeness means that an *honest* $\mathcal{P}$ who actually knows the witness $\mathbb{w}$ to a computation can always convince $\mathcal{V}$, while soundness refers to the property that a *malicious* $\mathcal{P}$ cannot convince $\mathcal{V}$ of a false statement except with negligible probability [30], [31].

B. How zk-STARK Proofs Work

Pythia employs the provably secure zk-STARK protocol [18] as its zero-knowledge back-end, which enables RAM-based ZKPs and comprises two phases: *arithmetization* and *low-degree testing*. The former refers to the reduction of a computational problem $\mathbb{A}$ into the algebraic problem of checking whether a certain polynomial has a low degree. The low-degree testing phase employs Reed-Solomon (RS) error-correction codes and RS proximity testing to show that the generated polynomial is actually of low degree [32]. Contrary to other ZK protocols, zk-STARK supports a universal trust-less setup, as we discuss in Section VII.

Arithmetization: $\mathcal{P}$ first expresses $\mathbb{A}$ as a sequence of T vectors that correspond to an *execution trace*. Each vector represents a single step of the computation, while the set of vectors compose $\mathbb{A}$. Then, for every two consecutive vectors in the execution trace, $\mathcal{P}$ and $\mathcal{V}$ agree on a set of assertions (expressed as polynomial constraints on field $\mathbb{F}$) that should hold during the execution (e.g., if two values are added, the result equals their sum). If any of these constraints are not valid, $\mathcal{V}$ would reject the computation since $\mathcal{P}$ either did not know the witness $\mathbb{w}$ or tried to cheat by tampering with the execution trace. In the last step of arithmetization, $\mathcal{P}$ uses $\mathbb{w}$ to compute a valid execution trace (T steps long) along with polynomial constraints and combines them by computing their linear combination to generate a single low-degree polynomial.

Low-degree testing: In this step, $\mathcal{P}$ convinces $\mathcal{V}$ that a given function $f : L \rightarrow \mathbb{F}$ is equal to a polynomial of degree smaller than a constant d. In other words, there exists a polynomial over $\mathbb{F}$ that *agrees* with f in the entire domain (i.e., $\forall x \in L : p(x) = f(x)$) and its degree is less than d. Typically, this problem requires querying the polynomial at every point in L since it might agree with f at every other point except one. $\mathcal{P}$'s goal is to enable $\mathcal{V}$ to query auxiliary functions on locations of $\mathcal{V}$'s choice, however, the verifier may not trust the prover to be honest. To solve this problem, $\mathcal{P}$ can use cryptographic commitments to functions of her choice [33], [34], and employ Merkle trees so that $\mathcal{V}$ can request $\mathcal{P}$ to reveal any of them. Commitments ensure that $\mathcal{P}$ will reveal the correct values she initially committed to and has not cheated. zk-STARK utilizes the *FRI protocol* (Fast Reed-Solomon Interactive Oracle Proofs of Proximity) that reduces the number of queries to be logarithmic to the size of L by recursively splitting the original polynomial into two polynomials of degree less than $d/2$ until $\mathcal{P}$ can send a constant value to $\mathcal{V}$ [32]. By proving that the function encoding the execution trace and the constraints is of low degree, $\mathcal{P}$ proves she knows a $\mathbb{w}$ so that the computation $\mathbb{A}(\mathbb{x}, \mathbb{w})$ outputs $\mathbb{y}$. To prevent leaking any information about $\mathbb{w}$, $\mathcal{P}$ selects a random value R so that $R \gg T$, generates a sequence of values $\{b_1, \ldots, b_{R-T}\}$, and computes the interpolating polynomial by combining this random sequence with the RAM's execution trace coefficients using Lagrange's interpolation method.

zk-STARKs in Pythia: In this work, we generalize the SHA-256 example (discussed in Section II-A) and implement a library of custom ZK modules that can evaluate both combinational and sequential circuits as the public computation $\mathbb{A}$. Specifically, each ZK module consumes a secret 3PIP netlist as the witness input $\mathbb{w}$ of $\mathbb{A}$, and a public input test vector $\mathbb{x}$ (provided by $\mathcal{V}$); in effect, algorithm $\mathbb{A}$ simulates the private netlist $\mathbb{w}$ on input $\mathbb{x}$. Each module is tailored to prove different aspects of the target circuit,

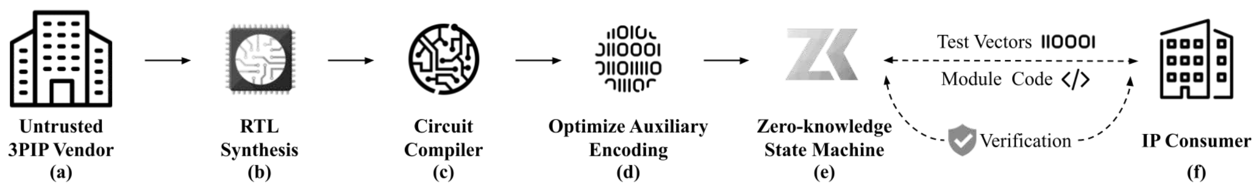


Fig. 1. **Overview of Pythia:** (a) The 3PIP vendor ($\mathcal{P}$) possesses an IP described in a Hardware Description Language. (b) $\mathcal{P}$ synthesizes the IP and generates a gate-level netlist. (c) $\mathcal{P}$ determines the evaluation order of the gates and transforms the IP into a zero-knowledge friendly encoding for ZK state machines. (d) $\mathcal{P}$ minimizes the number of intermediate wire values required to evaluate the netlist and divides the encoding into independent shares that can be evaluated in parallel. (e) The 3PIP vendor executes a module (e.g., functional, performance, area verification) with the circuit specification as private input and public test vectors chosen by the IP consumer. (f) The two parties interact and $\mathcal{P}$ convinces $\mathcal{V}$ about the computational integrity of the zero-knowledge evaluation.

such as functionality, max frequency, and estimated area/power. Pythia's evaluation output is a public vector $\mathbb{y}$ combined with a cryptographic proof of the correctness of the computation.

C. Lightweight Pseudorandom Functions with Extended Input

A deterministic function that can be efficiently computed on an input block and generate an output block that is computationally indistinguishable from an output generated by a truly random function is called a pseudorandom function (PRF) [35]. In practice, PRFs can be instantiated using secure block ciphers (such as Speck [36]), which combine a secret key with an input plaintext block to transform the plaintext into a (randomly looking) ciphertext block. Leveraging a universal hash function (UHF), the fixed input block size of a PRF can be extended to a secure PRF of arbitrary input size by applying a UHF operation before invoking the PRF (i.e., a $PRF(UHF(\cdot))$ composition) [37, Section 4.2].

Based on the blueprint of Carter and Wegman [38], a UHF $\mathcal{U}$ can be constructed as a polynomial of degree-ℓ modulo a prime number q that is evaluated at input point k. As discussed in [39], $\mathcal{U}$ can be defined over ℓ input blocks m_1 to m_ℓ (treated as polynomial coefficients) and a secret key k. In this case, $\mathcal{U}_k(m_1, \ldots, m_\ell) = m_1 k^\ell + m_2 k^{\ell-1} + \cdots + m_\ell k \bmod q$, with $m_i \in \mathbb{Z}_q$, is a lightweight UHF with a collision probability $\varepsilon \leq \ell/q$ for any pair of distinct inputs [40]. Using Horner's method, $\mathcal{U}$ can be computed iteratively and can be encrypted using the Speck cipher with key k' to construct an efficient, secure PRF with input extended over ℓ blocks. This PRF prevents adversaries from detecting if a collision has occurred and is secure against forgery attacks [41]. Alg. 1 computes the hash digest of message m with keys k, k'.

Algorithm 1 Lightweight Collision-Resistant Hash Function

```
Input: $k \in \mathbb{Z}_q$, $k' \in \mathbb{F}_{2^n}$, $m = (m_1, m_2, \ldots, m_\ell) \in \mathbb{Z}_q^\ell$
1: procedure $LCRHF_{k,k'}(m)$
2:     $\mathcal{H} \leftarrow m_1 \cdot k \bmod q$                    ▷ $q$ is a prime
3:     for $i \leftarrow 2$ to $\ell$ do                             ▷ Horner's method
4:         $\mathcal{H} \leftarrow k \cdot (\mathcal{H} + m_i) \bmod q$      ▷ $\ell - 1$ iterations
5:     return SPECKENCRYPT$_{k'}(\mathcal{H})$                        ▷ Speck alg. [36]
```

D. Threat Model

In this Section, we elaborate on the different threat scenarios we consider in our approach to address the problem of mutual mistrust between 3PIP vendors and system integrators. Our threat model is twofold: on one hand, we assume a cheating prover $\mathcal{P}$ that does not possess an IP and attempts to trick an honest verifier $\mathcal{V}$, and on the other hand, we assume a cheating verifier $\mathcal{V}$ that tries to learn any information from the communication with $\mathcal{P}$ about the IP.

Cheating Prover: A cheating $\mathcal{P}$ is an adversary with access to the capabilities of a 3PIP vendor and has incentives to deceive the system integrator ($\mathcal{V}$) while attempting to sell an IP that does not meet the agreed-upon functionality. The buyer (system integrator) is expected to test the IP using multiple input vectors and verify the correctness of the outputs, along with the attached ZK proof. The adversary succeeds if she produces a false proof that will convince $\mathcal{V}$ to accept it. Pythia features a configurable security parameter λ that defines the *soundness error* as $\epsilon = 2^{-\lambda}$ and determines the probability that an adversary can successfully deceive an honest $\mathcal{V}$ in the above interactions. Minimizing ϵ is possible by increasing the interaction rounds between $\mathcal{P}$ and $\mathcal{V}$ [18], [32].

Cheating Verifier: Here, a malicious $\mathcal{V}$ is assumed to behave without restrictions and not necessarily follow the protocol specification in order to extract any information about the secret IP. For example, a cheating buyer (system integrator) may attempt to learn the IP netlist before paying, even though the vendor ($\mathcal{P}$) does not wish to disclose the netlist until after payment is received. If $\mathcal{P}$ follows the protocol faithfully, $\mathcal{V}$ will never learn anything about the private witness $\mathbb{w}$ from interacting with $\mathcal{P}$, except the fact that $\mathbb{A}(\mathbb{x}, \mathbb{w})$ is true, which is guaranteed by the underlying ZK protocol. Lastly, we don't consider trivial cases where $\mathcal{V}$ completely withdraws from the protocol between the two parties (e.g., by ignoring messages).

III. The Pythia framework

A. Overview of Pythia

In this work, we present the Pythia framework: a novel method for privacy-preserving 3PIP verification. Pythia utilizes zero-knowledge proofs to enable IP vendors to prove to IP consumers that they possess a 3PIP with some agreed-upon functional specifications without revealing its design. In addition to functional verification, Pythia enables proving in zero-knowledge various circuit properties, such as estimated performance, area, and power. To that end, we have designed a library of ZK state machine modules where each one consumes a netlist as private input and proves a different property of the circuit. An overview of our framework is depicted in Fig. 1 and discussed in the following sections.

1) Privacy-Preserving Functional Verification: The fundamental idea of the Pythia framework is to enable 3PIP vendors (who act as $\mathcal{P}$) to prove to system integrators (i.e., $\mathcal{V}$) that *they possess an IP* with some predetermined specifications. In the first step of our methodology, depicted in Fig. 1(a), the 3PIP vendor synthesizes an IP described in a Hardware Description Language (HDL), such as Verilog, and creates a gate-level netlist. Consecutively, $\mathcal{P}$ uses the circuit compiler of Pythia to determine the evaluation order of the gates and resolve any inter-dependencies between gate inputs and outputs from previous gates, before finally converting the IP into a ZK-friendly format that can be evaluated by Pythia's ZK state machine. In the next step, $\mathcal{P}$ utilizes Pythia's optimizer to minimize the number of intermediate wires required to evaluate the circuit by employing bit-packing and graph coloring techniques. The optimizer also splits the netlist into multiple parallel shares that can be evaluated independently in ZK. Then, $\mathcal{P}$ and $\mathcal{V}$ agree on the algorithm $\mathbb{A}$ that $\mathcal{P}$ will execute (i.e., a state machine that evaluates circuits), and the verifier provides test vectors for the IP that are encoded as public inputs $\mathbb{x}$. Finally, $\mathcal{P}$ executes $\mathbb{A}$ locally to simulate the private netlist (input $\mathbb{w}$) using the public test vector (input $\mathbb{x}$) and computes a public output $\mathbb{y}$ that is sent to $\mathcal{V}$ along with a secure cryptographic proof that the circuit simulation was carried out faithfully.

2) Library of ZK Modules: Besides functional verification, Pythia supports various algorithms as zero-knowledge state machines, dubbed *ZK modules*, that can be used to prove different properties of the private netlist without revealing any details about it. More specifically, we implemented a library of modules that can assess (a) the expected performance of a circuit by estimating its critical path, (b) the area of a circuit by analyzing the different types and numbers of gates, and (c) the estimated power consumption based on the gate switching activity. Each module is implemented as a different zero-knowledge computation (i.e., as a different $\mathbb{A}$ algorithm) that can be executed within Pythia. Notably, in all aforementioned modules, the input IP netlist always remains hidden from $\mathcal{V}$.

3) Pythia back-end: The back-end of Pythia utilizes the zk-STARK protocol [18] to argue about the computational integrity of its state machine that simulates a circuit netlist and verifies different properties of it. Internally, Pythia leverages zk-STARK's programming interface to implement the arithmetization procedure and enable expressing a computation as a sequence of state machine transitions along with polynomial constraints that should hold during that computation. Then, Pythia employs these constraints to prove the correctness of the execution and all transitions of the state machine.

In our approach, we implement a circuit simulator as a state machine, and its state vector is initialized with all zeros. Each step of the circuit simulation algorithm copies the previous state vector, modifies at most one value of it, and appends it after the previous state vector. The generated sequence of state vectors comprises a two-dimensional table that represents the computation (i.e., the *execution trace* in Section II-B). Each new state vector can be updated in a variety of different ways, each of which corresponds to the desired operation that is performed in zero-knowledge. These operations can be arithmetic such as addition and multiplication, bitwise (i.e., conjunction, disjunction, etc.), comparisons, as well as operations that affect the control flow (e.g., a multiplexer). As discussed, our state machine supports two different types of inputs, namely *public* (for the test vectors that $\mathcal{V}$ provides), and *private* that correspond to the witness (i.e., the netlist that only $\mathcal{P}$ knows). Leveraging the security guarantees of zk-STARK [18] in our back-end, Pythia asserts the validity of each transition in the execution trace (each corresponding to a state machine transition) by imposing polynomial constraints and asserts their satisfiability to $\mathcal{V}$. In effect, Pythia achieves the *seemingly impossible task* of convincing a system integrator about the functionality and various properties of an IP without ever revealing its composition.

B. From IP Netlists to ZK-friendly Encoding

Pythia enables automatic compilation of IPs described as HDL programs into a zero knowledge-friendly format that can be interpreted and utilized by our back-end.

1) RTL Synthesis: As illustrated in Fig. 1b, Pythia synthesizes HDL (e.g., Verilog) programs into netlists consisting of logic gates and primitive memory structures like flip-flops. Without loss of generality, Pythia employs the Yosys Open SYnthesis Suite framework that performs RTL synthesis on an IP implemented in Verilog and generates the corresponding netlist in the Electronic Design Interchange Format (EDIF) [42]. During RTL synthesis, Pythia instructs Yosys to perform various optimizations, like removing unused wires and mapping cells to standard logic gates and small multiplexers.

2) Combinational Circuits: The generated EDIF netlist is provided as an input to our compiler (Fig. 1c), which parses the circuit and identifies all of its gates and wires. Our compiler then performs a second pass in which it associates each gate with specific input and output wires, and uses this information to determine the correct evaluation order of the gates, and eliminate any dependencies between intermediate gate inputs and upstream gate outputs by creating a directed acyclic graph (DAG). Pythia's DAG determines the evaluation order of the circuit's gates by tracking which gates precede each specific gate and then running a topological sort that resolves all of the dependencies in the netlist. After eliminating all dependencies, our compiler serializes the circuit so that Pythia's ZK state machine can prove all gates in sequence. Our compiler also transforms any gates of the circuit that take more than two inputs into a combination of two-input gates; for instance, a three-input `AND` gate is transformed into a pair of two-input `AND` gates. Finally, Pythia assigns a gate identifier to each logic gate (i.e., `AND`, `OR`, `XOR`, etc.) and writes the encoded IP to a file that is used as witness input `w` by our ZK modules.

3) Sequential Circuits: Evaluating sequential circuits can be achieved similarly; however, due to complex structures such as the clock signal and flip-flops, they involve a more complicated encoding than combinational circuits. In a software simulation setting where each gate operation amounts to a function call and cannot be "re-used" in the same way that hardware can, sequential evaluation is non-intuitive. The first and most important problem is to deal with the clock. We address this by effectively "unrolling" the circuit for each clock cycle and re-evaluating the gates when the clock ticks. Specifically, *on the first clock cycle* we fully evaluate the circuit and when a flip-flop is encountered, we set its output to 0 and buffer the input. At the start of the next clock cycle, the buffered input to the flip-flop is propagated to the output and the circuit is re-evaluated. We observe, however, that this method of re-executing the entire circuit on each clock cycle remains inefficient since many gate outputs remain unchanged between consecutive clock cycles. To account for this, Pythia re-evaluates *only the gates that depend on upstream flip-flops* in the dependency graph. This effectively means that if a flip-flop is encountered on the same path as any given gate in the circuit, this gate is re-evaluated for each clock cycle. This strategy ensures that any gates not connected to and influenced by a sequential circuit element are not needlessly re-computed.

C. Zero-Knowledge Circuit Evaluation

The fundamental operation of Pythia is the evaluation of Boolean circuits by executing a zero-knowledge state machine on any input test vector while preserving the privacy of the actual netlist. At a high level, Pythia first evaluates all these private gates and also proves the integrity of the computation. The initial state of our ZK machine corresponds to a vector filled with zeros and the state transitions denote the evaluation of logic gates. In each step, Pythia reads a logic gate from the private input, and depending on the type of gate, it determines what operation to perform on the two input wires to compute the correct output. The first level of gates (i.e., the gates with no dependencies) read circuit inputs directly,

Algorithm 2 State Machine for Circuit Evaluation

Input: Circuit $\mathcal{C}$ netlist (private), test-vector (public)

```
1:  procedure EvaluateCircuit
2:      $\mathcal{H} \leftarrow 0$                                    ▷ Keeps track of the LCRHF of the IP
3:      for $idx \in \mathcal{C}_{primary-inputs}$ do
4:          read $t$ from test-vector                                 ▷ Test vector input bit $0/1$
5:          $\vec{S}[idx] \leftarrow t$                               ▷ Initialize the state vector ($\vec{S}$)
6:      for $gate_{ID}, in_1, in_2, out \in \mathcal{C}$ do
7:          $\mathcal{H} \leftarrow LCRHF_{k,k'}(\mathcal{H}, gate_{ID}, in_1, in_2, out)$    ▷ Alg. 1
8:          if $gate_{ID} = AND$ then
9:              $\vec{S}[out] \leftarrow \vec{S}[in_1] \wedge \vec{S}[in_2]$
10:         else if $gate_{ID} = XOR$ then
11:             $\vec{S}[out] \leftarrow \vec{S}[in_1] \oplus \vec{S}[in_2]$
12:         else if $gate_{ID} = \dots$ then $\dots$                  ▷ NAND, NOR etc.
13:     return $\vec{S}, \mathcal{H}$
```

while intermediate gates depend on the outputs of preceding gates; to evaluate such gates, Pythia further identifies where to read their inputs from. Therefore, after each gate evaluation, we store the output at a *pre-determined index location* of the state vector, which is also encoded in the private input.

Gate evaluation: For each gate evaluation, Pythia encodes the gate identifier and three state vector indices: two for the gate input values and one for the output of the gate. Moreover, all test-vectors and the state machine algorithm $\mathbb{A}$ that evaluates Boolean circuits are public so that both $\mathcal{P}$ and $\mathcal{V}$ can access them; conversely, the gates and the indices used to evaluate each one are only known to $\mathcal{P}$. Although $\mathcal{V}$ is oblivious to the target 3PIP netlist she verifies, $\mathcal{V}$ is convinced that $\mathcal{P}$ has correctly evaluated a circuit using the public input vector, and generated the given public result.

Test vectors: System integrators would typically send many different input vectors to verify that the 3PIP claimed by $\mathcal{P}$ satisfies the predetermined functional specifications. Nevertheless, since our methodology hides the 3PIP from $\mathcal{V}$ by design, the verifier cannot simply trust that $\mathcal{P}$ did not alter the target netlist across evaluations with different test vectors. In other words, a malicious $\mathcal{P}$ could trick an honest $\mathcal{V}$ by using a different netlist each time, in an effort to prove a single 3PIP satisfies the required properties (across all test vectors), when such 3PIP may not actually exist. This is a crucial concern that would violate our threat model (Section II-D). To eliminate any such risks for $\mathcal{V}$, while still keeping the 3PIP private, Pythia employs a secure PRF to generate *authenticated digests* from the circuit during each evaluation.

Effectively, this proves to $\mathcal{V}$ that the same exact IP is used across all evaluations with different test vectors. Moreover, since proving the computation of a PRF in zero-knowledge may increase the execution overhead, we employ a lightweight PRF to minimize this impact on circuit evaluation; Pythia employs the PRF described in Alg. 1 with k, k' pre-shared between $\mathcal{P}$ and $\mathcal{V}$.

Circuit evaluation: Alg. 2 outlines the state machine used to prove functional properties of any circuit while evaluating it in zero-knowledge. In line 2, we initialize the hash digest $\mathcal{H}$ that Pythia generates during the evaluation of the 3PIP, while in lines 3 and 4 the state vector ($\vec{S}$), which stores the value of every wire in the netlist, is initialized with the public test vectors selected by $\mathcal{V}$. Then, the state machine iterates until it evaluates every private gate: First, Pythia reads the gate identifier along with input and output indices (line 6), and updates the hash digest using the PRF from Alg. 1 (line 7). Next, depending on the type of gate, the state machine carries a different operation on the inputs and updates the state vector at each out index with the computed output. At the end of the evaluation, Pythia returns the resulting state vector along with the hash digest of the IP as computed by the PRF.

IV. Library of Modules

Using our state machine for circuit evaluation as a backbone (i.e., Alg. 2), we design a library of auxiliary (AUX) modules that can prove in zero-knowledge various properties of the target 3PIP. In this section, we elaborate on three such modules that focus on area, power, and performance verification, without revealing any details about the corresponding circuit. As discussed in Section III-B, Pythia performs various optimizations during RTL synthesis to remove redundant blocks or unused wires, as well as transform any gates of the 3PIP that have more than two inputs into a combination of two-input gates. Thus, all AUX modules operate on technology-independent reduced netlists.

An important observation is that all our modules process all the gates of the 3PIP regardless of the inputs, resulting in a constant time execution across different test-vectors. To prove that the same 3PIP was used as private input for functional verification across all AUX modules, Pythia computes a secure hash digest of the 3PIP netlist while it parses it. This hash can then be compared against the hash computed by the circuit evaluation module of Pythia, proving that $\mathcal{P}$ did not switch the 3PIP across the different AUX modules.

A. Area Verification Module

The first AUX module we propose is an area verification state machine. In computer-aided design (CAD) context, area complexity typically refers to the problem of estimating the minimum number of gates required to implement a Boolean function by only having access to the high-level description of

Algorithm 3 Module to Prove Gate Distribution

Input: Circuit $\mathcal{C}$ netlist (private)

```
1: procedure AREAMODULE
2:     $\mathcal{H} \leftarrow 0$                                          ▷ Keeps track of the LCRHF of the IP
3:     $\vec{G} \leftarrow [0, \ldots, 0]$                                 ▷ Initialize gate vector ($\vec{G}$) with zeros
4:     for $gate_{ID}, in_1, in_2, out \in \mathcal{C}$ do
5:         $\mathcal{H} \leftarrow LCRHF_{k,k'}(\mathcal{H}, gate_{ID}, in_1, in_2, out)$     ▷ Alg. 1
6:         $\vec{G}[gate_{ID}] \leftarrow \vec{G}[gate_{ID}] + 1$          ▷ Gate type counter
7:     return $\vec{G}, \mathcal{H}$
```

the function [43], [44]. Therefore, previous works have focused on developing different techniques to estimate the area before implementing a design. Nevertheless, our ZK use-case is somewhat different since $\mathcal{P}$ has access to both the HDL description and the EDIF netlist of the IP. In particular, our approach focuses on proving to system integrators that $\mathcal{P}$ owns a 3PIP that meets some functional specifications, and on top of that, it satisfies certain area constraints.

In Pythia, we calculate the area of a circuit by tracking the different gate types and the cardinality of each gate. Our starting point is the circuit evaluation process discussed in Section III-C. Our area verification module, summarized in Alg. 3, utilizes different indices in a gate vector ($\vec{G}$) to track the number of flip-flops (FFs) and each gate type comprising the private 3PIP netlist (e.g., `AND`, `OR`, `NOT`, `XOR`, `NAND`, `NOR`, and `XNOR`). Each time a gate or FF is read from the private input, Pythia increases the corresponding $\vec{G}$ counter by one and then continues to the next. When all gates and FFs have been evaluated, Pythia outputs the hash digest and the $\vec{G}$ value. The former is used to prove that the same private 3PIP was used in this module as in the functional verification case, while the latter contains the values of all counters that can estimate circuit area and track the gate type distribution.

B. Performance Verification Module

The second property we can prove with Pythia using static analysis is the performance of a circuit by calculating the *path logical effort* [45] between inputs and outputs and locating the critical path of a circuit, which is the path with the longest delay. Different gates introduce different delays: for instance, `AND` and `OR` gates have approximately the same delay, whereas `XOR` gates incur higher delays since they may be constructed by a combination of other basic gates [46]. All gate delays are expressed in terms of a basic delay unit τ, which is the delay of an ideal fan-out-of-1 inverter with no parasitic capacitance. We formalize the absolute gate delay as the product of the normalized delay of the gate d and the basic delay unit τ. The normalized delay of an individual gate can be broken down into the summation of

TABLE I

LOGICAL EFFORT AND PARASITIC DELAYS OF COMMON LOGIC GATES.

Logic Gate	**Num. of Inputs**	**Logical Effort** g	**Parasitic Delay** p
`NOT`	1	1	1
`NAND`	2	$4/3$	2
`NOR`	2	$5/3$	2
`AND`	2	$7/3$	3
`OR`	2	$8/3$	3
`XOR/XNOR`	2	4	4

the delay of the gate that is dependent on the load f (i.e., *stage effort*), and the delay when the gate is not driving any load p (i.e., *parasitic delay*). Additionally, the stage effort f consists of logical effort g and electrical effort h (i.e., the *fan-out*), such that $f = g \cdot h$. The logical effort is the ratio of the input capacitance of a given gate to that of an inverter capable of delivering the same output current, while the fan-out is the ratio of the input capacitance of the load to that of the gate. Thus, the normalized delay of a single gate can be calculated by the following equation:

$$d = g \cdot h + p. \tag{1}$$

We summarize the parasitic delay and stage effort of different logic gates in Table I. Notably, these delays are part of our AUX module configuration and Pythia can easily substitute them with new delays in order to model a specific technology.

In Pythia, we implement the path logical effort methodology as introduced in [45], which is an extension of the aforementioned method for computing the delays of single gates. Given a path, the path logical effort G equals the product of the logical efforts of all the logic gates along the path (i.e., $G = \prod g_i$), whereas the path electrical effort H is the ratio of the capacitance of the last logic gate of the path to the input capacitance of the first gate in the path (i.e., $H = C_{out(path)}/C_{in(path)}$). We also calculate the path branching effort B as the product of the branching efforts at each of the stages along the path. The path effort F is defined as the product of the logical, electrical, and branching efforts of the path (i.e., $F = GBH$) and is used to minimize the delay of a certain path; the latter is minimized when the stage effort in each stage along the path is the same. Finally, the minimum delay achievable along the path is computed by

$$\hat{D} = NF^{1/N} + P, \tag{2}$$

where P is the path parasitic delay and equals the sum of the parasitic delays of each gate in the path (i.e., $P = \sum p_i$).

Our Heuristic Method: Minimizing the delay for every single path using the path electrical effort method incurs exponential overheads due to the number of input to output path combinations. For large circuits, this approach does not scale gracefully, which motivates the need for tailored optimizations in the context of ZK performance verification. Therefore, we introduce *a novel and efficient heuristic* that uses the single gate logical effort method (i.e., Eq. 1) to cascade the delays of the gates of any netlist in a single pass of the circuit and to identify the critical path. In particular, for each gate on a path, our heuristic method propagates the maximum upstream delay of both gate inputs and adds the normalized delay d of that gate. Since a forward parsing[1] of the netlist does not allow computing the exact electrical efforts h of each individual gate, our heuristic assumes that h is an integer equal to the fan-out of the gate (i.e., number of output wires). This assumption significantly reduces the computation cost and makes this method a heuristic; our ISCAS experiments demonstrate the high accuracy of our heuristic, as it can always correctly identify the critical path, while the heuristically computed path delay incurs less than 5% error in all cases.

After we identify the critical path, we apply the exact path logical effort method (i.e., Eq. 2) solely to the critical path. Pythia optimizes the aforementioned algorithm to compute the path logical effort of a path by updating all necessary parameters for Eq. 2 as we compute the heuristic delay, *requiring only a single pass*. Notably, applying our heuristic method to all paths (Alg. 4, black text only) uses significantly fewer state indices than the exact method of Eq. 2 (Alg. 4, both black and blue text), which effectively reduces the number of zero-knowledge operations required by our AUX module.

Exact Method: Alg. 4 summarizes our ZK algorithm $\mathbb{A}$ for heuristically identifying the critical path and calculating the exact path logical effort based on Eq. 2 and the formulas from Table I. The exact delay computation is highlighted in blue color and complements the heuristic algorithm (i.e., the black text). Specifically, Pythia processes the gates iteratively (line 9) and stores the heuristic delay in vector $\vec{D}$ (line 20), based on the propagated maximum delay of the two input wires (lines 15 and 16). For exact delay computation, Pythia keeps track of: (a) the path logical effort G by multiplying the logical efforts of single gates along the path (line 21), (b) the path parasitic effort P by adding the parasitics of single gates in the path (line 22), (c) the number of stages in the path N (line 17), and (d) the path branching effort B (line 18). We initialize five vectors to track D, G, B, P, and N with unique counters for every

[1]We always analyze the netlist in the same way as with the other modules (i.e., using a forward-pass), to prove the same netlist digest $\mathcal{H}$ is computed.

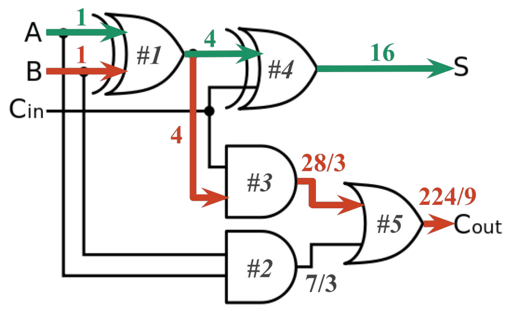
Algorithm 4 Module to Compute Path Logical Effort

Input: Circuit $\mathcal{C}$ netlist (private), Path electr. effort H (public)

```
1: procedure PerformanceModule
2:   $\mathcal{H} \leftarrow 0$                                         ▷ Keeps track of the LCRHF of the IP
3:   $\vec{D} \leftarrow [0, \ldots, 0]$                                ▷ Initialize the heuristic delay vector
4:   $\vec{G}, \vec{B} \leftarrow [1, \ldots, 1]$                       ▷ Path logical and branching effort vectors
5:   $\vec{P}, \vec{N} \leftarrow [0, \ldots, 0]$                       ▷ Path parasitic delay and gate counter vectors
6:   for $idx \in \mathcal{C}_{primary-inputs}$ do
7:     read $b$ from $\mathcal{C}$                                      ▷ Read branching effort value $b$
8:     $\vec{h}[idx] \leftarrow b$                                      ▷ Initialize the fanout vector ($\vec{h}$) for primary wires
9:   for $gate_{ID}, in_1, in_2, out, b \in \mathcal{C}$ do
10:    $\mathcal{H} \leftarrow LCRHF_{k,k'}(\mathcal{H}, gate_{ID}, in_1, in_2, out)$   ▷ Alg. 1
11:    $\vec{h}[in_1] \leftarrow \vec{h}[in_1] - 1$                     ▷ $in_1$ is used, decrease $b$ for wire 1
12:    $\vec{h}[in_2] \leftarrow \vec{h}[in_2] - 1$                     ▷ $in_2$ is used, decrease $b$ for wire 2
13:    assert $\vec{h}[out] = 0$                                        ▷ Assert that $b$ at index $out$ was correct
14:    $\vec{h}[out] \leftarrow b$                                      ▷ Update $b$ for current gate at index $out$
15:    if $\vec{D}[in_1] > \vec{D}[in_2]$ then $max \leftarrow in_1$    ▷ Set $max$ to
16:    else $max \leftarrow in_2$                                       ▷ the wire with the maximum heuristic delay
17:    $\vec{N}[out] \leftarrow \vec{N}[max] + 1$                       ▷ Count gates in critical path
18:    $\vec{B}[out] \leftarrow \vec{B}[max] * b$                       ▷ Update path branching effort
19:    if $gate_{ID} = AND$ then
20:      $\vec{D}[out] \leftarrow \vec{D}[max] + \frac{7b}{3} + 3$      ▷ Update heuristic delay
21:      $\vec{G}[out] \leftarrow \vec{G}[max] * 7/3$                   ▷ Update path logical effort
22:      $\vec{P}[out] \leftarrow \vec{P}[max] + 3$                     ▷ Update path parasitic delay
23:    else if $gate_{ID} = \ldots$ then $\ldots$                      ▷ cf. Table I for NAND etc.
24:  $idx \leftarrow argmax(\vec{D})$                                   ▷ Find the index of the maximum value in $\vec{D}$
25:  return $\vec{G}[idx] * \vec{B}[idx] * H, \vec{N}[idx], \vec{P}[idx], \vec{D}[idx], \mathcal{H}$
```

logic gate (lines 3–6). After Pythia has processed all netlist gates, we find the index with the maximum heuristic delay and use it to return the exact path effort $F = GBH$ along with the path parasitic delay and the number of gates in the path.

To compute the path branching effort B, we augment the serialized 3PIP netlist with the branching information provided by the Pythia compiler. This creates a unique challenge that we have to address:



	G	B	P	N
Initial	1	1	0	0
Gate #1	4; 4	2	4	1
Gate #2	7/3	1	3	1
Gate #3	28/3	2	7	2
Gate #4	16	2	8	2
Gate #5	224/9	2	10	3
Critical	**24.9**	**2**	**10**	**3**

Fig. 2. **Path logical effort calculation:** The table depicts how the path logical effort G, path branching effort B, path parasitic delay P, and the number of logic stages in the path N counters are used to estimate the delay of any circuit using the gate delays from Table I. G, B, P, and N are updated based on the type of logic gate and the formulas in lines 17–22 in Alg. 4.

"How can we prove that the branching effort that was read from the serialized encoding is correct?". To convince $\mathcal{V}$ about this argument in ZK, Pythia tracks the branching effort of each gate in a separate state vector ($\vec{h}$ in Alg. 4) and decrements the corresponding index every time the output of one gate is used as input by another (lines 11 and 12). In lines 6–8 we read the branching effort of all input wires and initialize $\vec{h}$. When a new gate is processed, we read the branching effort (b) for this gate and check that the previous branching effort stored at that index is indeed zero (line 13). Finally, we update the correct index with the new branching effort (line 14). Alg. 4 proves the correct computation of G, B, P, and N to $\mathcal{V}$, who in turn can compute the path effort F based on the path electrical effort H, which yields the exact path delay $\hat{D}$ using Eq. 2. In addition to computing the critical path delay, this module uses the *private netlist input* to construct a hash digest of the 3PIP, so that $\mathcal{V}$ can ensure this hash equals those generated during functional verification and area verification (i.e., the 3PIP was not switched by $\mathcal{P}$).

Example: In Fig. 2 we demonstrate how Pythia computes $\vec{G}$, $\vec{B}$, $\vec{P}$, and $\vec{N}$ based on the two inputs and the delays from Table I. On the left-hand side of Fig. 2, we illustrate how $\vec{G}$ is calculated: Both input counters of gate *#1* have an initial value of 1 and the output counter gets the value 4 since it is an `XOR` gate. Similarly, the branching effort and parasitic delay of gate *#1* are 2 and 4 respectively. Lastly, N tracks the maximum number of gates along each path. Iterating in a similar manner with the rest of the gates, our method computes the critical path with $G = 24.9$, $B = 2$, $P = 10$ and $N = 3$. The path delays are then computed using the load capacitance H that $\mathcal{V}$ chooses.

Net Delay Estimation: Our aforementioned performance estimation method is based on circuit netlist IPs, so it does not incorporate net delays that depend on placement and routing; as with the standard logical effort method, our module assumes negligible wire capacitance and RC delay. Nevertheless, since interconnection delays are becoming more and more impactful [47], Pythia provides a heuristic to approximate the net delay of the critical path by taking into account the total number of wires in the path as well as an optional distribution of wire delays. The latter is agreed between $\mathcal{P}$ and $\mathcal{V}$ and can be derived from past designs of similar size and technology. If no distribution is provided, we assume that all wires in the critical path are "short" and have equal length. In this case, we adjust Alg. 4 output by adding the path net delays based on the average delay per wire, as agreed by $\mathcal{P}$ and $\mathcal{V}$.

C. Power Verification Module

Pythia also incorporates an AUX module for estimating the dynamic power consumption of a netlist to convince system integrators. Apart from *dynamic* power dissipation, circuits also draw *static* power. The former consists of the switching power (i.e., charging and discharging load capacitances as gate outputs change), while the latter is consumed through various circuit nodes and transistor leakages when a chip is not switching. In this module, we focus on calculating the gate switching activity of 3PIPs, since the static power dissipation is negligible compared to the dynamic power consumption. Various previous works (e.g., [48], [49]) propose different models to estimate the dynamic power consumption of netlists, and the core idea in all of these techniques is to predict the switching activity of the transistors in the circuits. Specifically, the dynamic power of a circuit can be expressed as a function of the probability that a gate transitions from 0 to 1, called *activity factor* α, and the clock frequency f so that $P_{dynamic} = \alpha \cdot C \cdot V_{DD}^2 \cdot f$, where C is the load capacitance and V_{DD} is the positive voltage [50].

Gate switching highly depends on the inputs and thus, the activity factor provides a way to estimate the probability that a gate transitions from 0 to 1 (the reverse transition does not consume any power [50, 5.1.2]), without exhaustively trying different input-vectors, which in many cases is impossible. Since the

TABLE II
LOGIC GATES SWITCHING PROBABILITIES: P_i DENOTES THAT NODE i IS 1 BASED ON INPUT A AND B PROBABILITY BEING 1.

Gate	**Probability** P_i
`NOT`	$\overline{P}_A$
`NAND`	$1 - P_A \cdot P_B$
`NOR`	$\overline{P}_A \cdot \overline{P}_B$
`AND`	$P_A \cdot P_B$
`OR`	$1 - \overline{P}_A \cdot \overline{P}_B$
`XOR`	$P_A \cdot \overline{P}_B + \overline{P}_A \cdot P_B$
`XNOR`	$1 - (P_A \cdot \overline{P}_B + \overline{P}_A \cdot P_B)$

activity factor depends on the logic function, in Table II we provide the probabilities P_i that the output of gate i is 1, based on the probabilities of its inputs A and B being 1. We also use $\overline{P}_i = 1 - P_i$ to denote the complement probability (i.e., node i is 0). Using the probabilities of each node, we can derive each activity factor as $\alpha_i = P_i \cdot \overline{P}_i$.

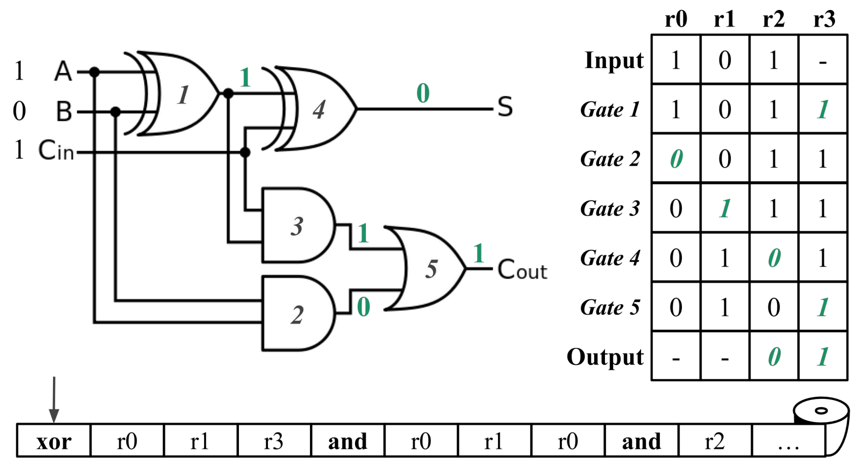
Pythia's power module processes the private 3PIP in a similar manner as in the critical path calculation, yet instead of tracking the different delays, we calculate the activity factor of each gate. All primary input probabilities are extracted from the inputs that $\mathcal{V}$ has provided, and then depending on the gate that Pythia reads, we compute the activity factor of each gate based on the formulas from Table II. For instance, if the input probabilities in Fig. 2 are $P_A = P_B = P_{C_{in}} = 0.5$, Pythia estimates the dynamic power consumption of the circuit as follows: The probability that *Gate #1* (i.e., XOR) outputs 1 is calculated as $P_1 = P_A \cdot \overline{P}_B + \overline{P}_A \cdot P_B = 0.5$, yielding an activity factor $\alpha_1 = 0.5 \cdot 0.5 = 0.25$. Then, Pythia processes *Gates #2* and *#3* (i.e., ANDs) and computes $P_2 = P_B \cdot P_A = 0.25$ and $P_3 = P_{C_{in}} \cdot P_1 = 0.25$ which result in $\alpha_2 = \alpha_3 = 0.25 \cdot 0.75 = 0.1875$. Likewise, $P_4 = P_1 \cdot \overline{P}_{C_{in}} + \overline{P}_1 \cdot P_{C_{in}} = 0.5$, and $\alpha_4 = 0.25$, while the probability that *Gate #5* (i.e., OR) is 1 is computed as $P_5 = 1 - (\overline{P}_3 \cdot \overline{P}_2) = 0.4375$, producing an activity factor $\alpha_5 = 0.4375 \cdot 0.5625 = 0.246$. Instead of arbitrarily setting the input probabilities, we employ a probabilistic model that extracts the primary input probability distribution based on $\mathcal{V}$'s test-vectors. Essentially, the probability of each input is calculated as the average of a specific input across multiple test-vectors. As in all library modules, Pythia computes a secure hash of the private netlist in ZK, so the Verifier can compare it with the one calculated during functional verification to be convinced that the estimated power consumption was calculated over the same 3PIP.

V. Pythia's Optimizer

In this Section, we propose a series of optimizations to effectively reduce the size of the state vector, since a smaller state vector results in improved performance for both $\mathcal{P}$ and $\mathcal{V}$. Finally, we introduce a methodology to split the functional and property verification modules to exploit parallelization.

A. *Efficient Wire Placement Using Register Allocation*

Apart from the area verification module that uses a constant number of state indices to keep track of the numbers and types of gates for each different netlist, the other modules require more indices as the total number of gates in the netlist increases. For small circuits, assigning each wire to a unique state vector index is feasible, however, for bigger circuits the size of the state vector required to hold all of these wires grows significantly, impacting both the time required to generate a proof and the time to verify it. Therefore, we apply a tailored *register allocation technique* that utilizes graph coloring to minimize the number of unique indices needed to hold the intermediate wires of a circuit. We incorporate



	r0	r1	r2	r3
Input	1	0	1	-
Gate 1	1	0	1	*1*
Gate 2	*0*	0	1	1
Gate 3	0	*1*	1	1
Gate 4	0	1	*0*	1
Gate 5	0	1	0	*1*
Output	-	-	*0*	*1*

xor	r0	r1	r3	**and**	r0	r1	r0	**and**	r2	…

Fig. 3. **Adder evaluation:** The numbers on the gates denote the evaluation order, also illustrated by the row labels of the table (execution trace) on the right-hand side. The green values represent which variable changed after the evaluation of the gate denoted by the row number. The variables $r0$–$r3$ can be interpreted either as four separate indices or as one 4-bit block. The serialized private input encoding a part of the above circuit is depicted on the bottom.

this optimization in the Pythia compiler during the IP transformation phase: instead of assigning unique indices to each wire, Pythia deftly allocates and re-uses indices for the input, intermediate, and output wires in the state vector.

Fig. 3 demonstrates how the register allocation technique effectively reduces the number of unique indices and how they can be reused to prove the functionality of an adder. Without this optimization, the circuit in Fig. 3 would require 3 input (i.e., A, B, C_{in}), 2 output (i.e., S, C_{out}), and 3 intermediate wires (i.e., to hold the results of gates 1, 2 and 3), resulting in a total of 8 different indices. In Fig. 3 we demonstrate how our graph coloring technique allocates each wire to an index and enables Pythia to evaluate the adder with solely 4 indices. The table on the right-hand side of the figure depicts the execution trace of the adder (i.e., how the state changes with each gate evaluation). In the first row of the execution trace the state vector is initialized with the values of the primary inputs: $A = 1$, $B = 0$ and $C_{in} = 1$. In the second row, gate $\#1$ is evaluated and its output is written at the last index of the state (highlighted in green). Each row corresponds to the evaluation of the next gate, while the last row holds the two outputs of the adder. At the bottom of Fig. 3, we show a part of the 3PIP witness input `w` that the Pythia compiler generated from the adder circuit using register allocation. First, the state machine reads the `XOR` gate and the two input variables (`r0`, `r1`), then it evaluates the gate using the contents of the two indices (i.e., 0 and 1) and stores the result to the output variable (`r3` $= 1$). Then, our state machine continues to the next operation. This optimization significantly reduces the total number of required indices, and as we show later in Section VI-B, this is crucial for the performance of Pythia.

B. Bit-Packing

Large combinational and sequential circuits that consist of thousands of wires and gates require a considerable amount of indices to hold intermediate wire values even when register allocation has been applied. Register allocation techniques are not as effective in circuits with many gates at the same level (i.e., circuits with large width), since all these wires should hold their values at the same time. An important observation about our functional verification module is that each state index stores only a binary value, which yields state vectors with many indices. Notably, increasing the number of indices in the state vector impacts the number of zk-STARK operations, which can affect performance. Therefore, *to minimize the number of indices*, yet continue storing the same amount of information, we employ a *bit-packing* optimization that organizes multiple wires into multi-bit blocks under the same state vector index. For instance, using this bit-packing optimization, the state in Fig. 3 becomes a single 4-bit register R_0 instead of four 1-bit registers (r_0 to r_3). In this example, R_0 will transition as follows: $1011 \rightarrow 0011 \rightarrow \cdots \rightarrow 0101$.

Specifically, the bit-packing scheme of Pythia can utilize uniquely-indexed 64-bit blocks, which reduces the number required by a factor of 64. Notably, Pythia can still access individual bits within each 64-bit block. To enable this optimization, Pythia's compiler can encode both the block index within the state vector and the bit index within the block for the inputs and output of each gate. Therefore, in addition to the previous four inputs (i.e., $gate_{ID}, in_0, in_1, out$ in line 6 of Alg. 2), each gate evaluation includes three-bit indices as well (i.e., one for each input and one for the output). Although this feature seemingly complicates Pythia's state machine as it incurs bit-shift operations to read/write individual bits within a block, this overhead is negligible compared to the ZKP cost savings from having fewer state vector entries. This optimization is mostly applicable to functional verification, as the power and performance modules already store multiple bits in each index, while the area module uses a constant state vector size so bit-packing does not impact its performance.

C. Execution Parallelism

To further optimize the performance of our methodology, we augment Pythia's optimizer to enable parallelized proof generation and verification. Specifically, we observe that all four Pythia modules operate iteratively over a given netlist. The execution trace of each module is initialized with a zero state vector and transitions to different states during the computation; given any intermediate state and a private 3PIP, Pythia can always continue the computation and converge to the correct output. Therefore, we can break down the problem of proving one big execution trace into the problem of proving the faithful execution of two smaller execution traces, along with a provable transition between the two. We refer to each smaller

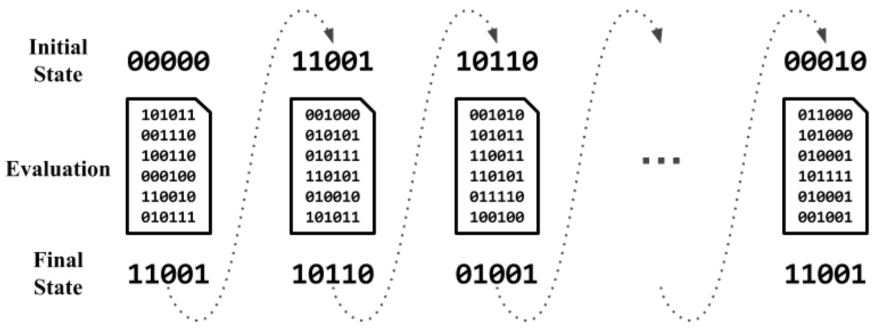


Fig. 4. Chaining execution to enable parallel verification. Pythia divides large executions into multiple shares and pre-computes the intermediate states locally. The simulator computes the PRF digest of the machine state and compares it with the digest provided in the public input to verify its integrity. Each share can be verified independently and in parallel.

execution trace that is a part of a bigger trace as a *share*. We demonstrate this concept in Fig. 4, where the sample initial state of the first share is 00000 and after the evaluation of the share its state becomes 11001. The latter initializes the second share, which will produce a new state 10110, and so on. The final state 11001 represents the result of the computation, which can be either the output wires of a netlist (functional verification), or different counters encoding the area, performance, or power estimations.

Although the shares in Fig. 4 are constructed honestly by the 3PIP vendor, $\mathcal{V}$ is not able to immediately verify if the intermediate states are initialized correctly or a malicious $\mathcal{P}$ has modified them. To convince $\mathcal{V}$ that these shares are actually parts of the original execution trace, and also preserve the confidentiality of the intermediate state vectors, we compute integrity measurements of each state vector using the lightweight PRF discussed in Section II-C. The 3PIP vendor calculates these publicly authenticated digests and shares them with $\mathcal{V}$; notably, only $\mathcal{P}$ knows the actual netlist data that produced each digest. At last, the system integrator verifies that all PRF digests at the end of each share are the same as the ones used at the beginning of each next share.

So far, we demonstrated how to decompose a big execution trace into smaller consecutive shares, and chain them in order to compute the correct output. Nevertheless, to exploit parallelism in zero-knowledge we also need to pre-compute the intermediate state vector values. In this case, Pythia quickly evaluates the netlist without creating a ZK proof, and computes the starting and ending state vectors of each share, along with their PRF digests. The offline execution overhead for $\mathcal{P}$ is negligible compared to the online proof generation timing. This method is not affected by any dependencies between consecutive shares, as all intermediate state vectors and their hash digests are precomputed, so proof generation of each share can take place in parallel. Finally, $\mathcal{V}$ can verify that the public PRF digests at the end of each share match the ones used to initialize the next share to ensure correctness.

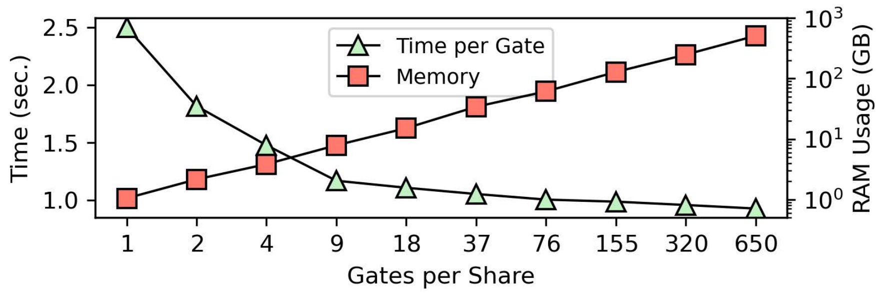


Fig. 5. Memory/Efficiency per gate trade-off for the prover. Horizontal axis shows the maximum number of gates per share that can be evaluated in less than a power of 2 state machine transitions.

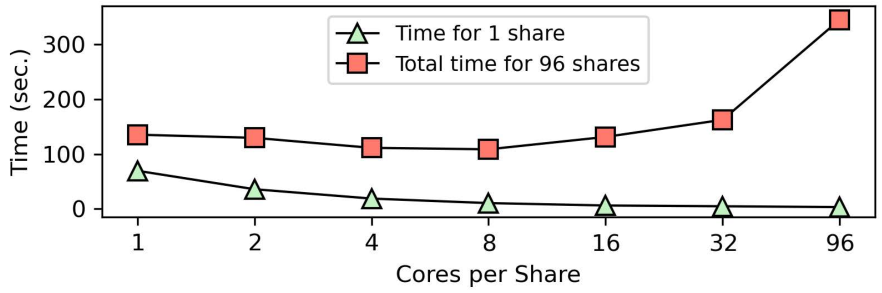


Fig. 6. Time measurements for proving 1 and 96 shares with a different number of threads per share. The red squares depict the timings for 1 core/share (prove all 96 in parallel) to 96 cores/share (prove shares sequentially).

VI. Experimental Results

A. Experimental Setup

In this Section, we evaluate the applicability and performance of our framework using selected benchmarks from the ISCAS'85, ISCAS'89, and ITC'99 suites. To verify the correctness of our circuit evaluation (Alg. 2), we employ a variety of input-output pairs for our benchmarks, as well as two arithmetic cores from [39]: an 8-bit high-radix sequential multiplier and a 12-bit sequential fast modular

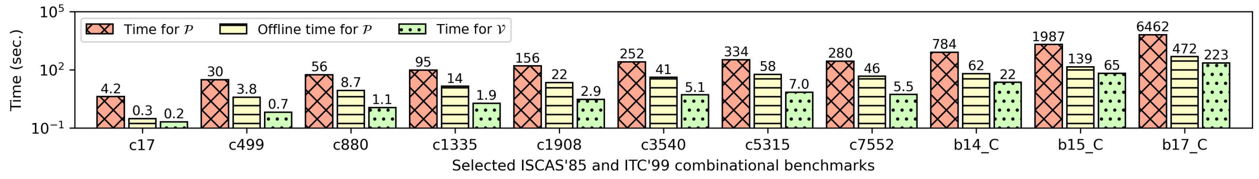


Fig. 7. $\mathcal{P}$ and $\mathcal{V}$ experimental results for selected benchmarks from the ISCAS'85 and ITC'99 suites.

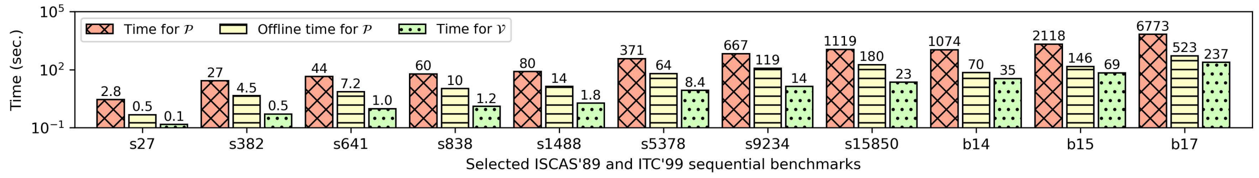


Fig. 8. Amortized $\mathcal{P}$ and $\mathcal{V}$ evaluation time per cycle (over 10 cycles) using selected benchmarks from the ISCAS'89 and ITC'99 suites.

reduction core for Mersenne prime moduli. We synthesize the benchmarks to produce EDIF netlists using the Yosys Open SYnthesis Suite framework [42]. We used Verilog and VHDL implementations for the ISCAS and ITC benchmarks, respectively. Yosys features various parameters that produce different netlists depending on the user's preferences (e.g., optimize for performance, space, etc.). In this work, we synthesized our benchmarks with the `proc` and `flatten` flags. Notably, more sophisticated tools may result in more optimized netlists, however, our goal in Pythia is to optimize the cost per gate for $\mathcal{P}$ and $\mathcal{V}$, rather than to optimize the benchmarks themselves.

The core of the Pythia framework is implemented in C++, while Pythia's compiler and optimizer are implemented in Python 3. To fully investigate parallelization, we obtained our experimental results on an m5.24xlarge AWS EC2 instance with two Intel Xeon Platinum 8175M cores running at 2.5 GHz and hyper-threading, resulting in 96 virtual cores and 748 GB RAM. The system is running Ubuntu 18.04 with the 4.15.0 Linux kernel and the g++ 7.4.0 compiler.

B. Performance Evaluation

The timing and memory overheads incurred by Pythia's back-end highly depend on both the length of the execution trace and the size of the state vector. As the execution trace becomes bigger, both the execution timing and the memory required for $\mathcal{P}$ increase. The verification timing is also affected by the same factors, however, it is always poly-logarithmic ($polylog(T)$) to the size of the execution trace T. An important property of Pythia's back-end (inherited from zk-STARK) is that the proving cost for any two execution traces whose size is less than the same power of 2 is approximately the same [18]. For instance, a trace with 942 steps incurs roughly the same timing and memory overheads as a trace that has 1020 steps since both have less than 2^{10} transitions.

1) Splitting Shares Trade-off: In this experiment, we study the optimal size for our shares by investigating the number of gates in each share. Increasing the number of shares allows proving more shares in parallel, however, the cost of calculating the PRF digest in every share will eventually dominate the proving time. Conversely, having fewer shares makes the cost for the PRF negligible compared to the total cost of proving the evaluation of each share, while the required memory increases as well. In Fig. 5 we illustrate the aforementioned trade-off based on the proving time per gate for our functional verification module: the *left* vertical axis shows the time per gate, while the *right* vertical axis indicates the required memory for each splitting configuration. While the time for $\mathcal{P}$ increases as the execution trace becomes longer, having more gates in each share reduces the proving time per gate. The green triangle trend in Fig. 5 indicates that increasing the number of gates per share (i.e., having fewer shares) decreases the overall proving time; however, the associated memory cost (red squares in Fig. 5) scales linearly with the number of gates per share. Therefore, the cost for $\mathcal{P}$ depends on the number of shares that can fit in the available memory of the target host. For our experimental setup, we identified that 9 gates per share strike a good trade-off between proving time and required memory.

2) Two levels of Parallelization: Pythia is *doubly* parallelizable: (a) using multiple independent shares, we can split the computation and evaluate the shares in parallel, and (b) Pythia's back-end utilizes multiple threads to parallelize the proof generation in each share. This motivates our next experiment that investigates how we can optimize parallelism in a multi-core host. The green triangles in Fig. 6 present the proving time for 1 share using a different number of cores. We observe that having more than 8 cores per share reduces the relative speedup (diminishing returns), so the cores per share should be balanced with the number of shares processed in parallel on all available cores.

Moreover, the red squares in Fig. 6 investigate the total cost for proving 96 shares by varying the number of allocated cores per share. On one extreme, we can assign 1 core per share in order to verify all 96 shares in parallel (i.e., leftmost red square in Fig. 6), while on the other extreme we can assign all 96 cores to verify 1 share at a time and repeat the process 96 times (rightmost red square). If we allocate 8 cores per share, we can verify 12 shares in parallel per iteration (note, we need 8 iterations to verify all 96 shares), our setup achieves the fastest overall time for $\mathcal{P}$, which further confirms the finding of the previous paragraph.

3) Discussion of Experimental Results: We evaluate Pythia using the ISCAS'85, ISCAS'89, and ITC'99 benchmark suites [51]–[53] with random input test-vectors (chosen by the IP consumer). Specifically, $\mathcal{V}$ is responsible for choosing multiple test-vectors to supply to Pythia and this selection does not impact the proving time. We observe that each benchmark's proving time depends on the total number of gates in the netlist, rather than the distribution of the input pattern, even though different inputs trigger different gates. In Figs. 7 and 8, we report our performance evaluation results for a selection of benchmarks, using a splitting of 9 gates per share and 8 cores allocated per share. In particular, Fig. 8 shows the amortized cost per cycle (over ten clock cycles) for each of our sequential benchmarks. As discussed in Section V-C, in order to prove multiple shares in parallel, $\mathcal{P}$ needs to quickly compute the intermediate state vectors offline (i.e., without generating a proof). Therefore, each benchmark in Figs. 7 and 8 reports the aforementioned offline cost (using the yellow bars in the middle), as well as the online costs for both $\mathcal{P}$ and $\mathcal{V}$ using the red (left) and green (right) bars, respectively. All timings depend on the length of the execution trace, which in turn depends on the number of gates in each netlist share; in this case, all shares comprise exactly 9 gates, so the trace length is constant. Thus, the factor that dominates Pythia's performance is the number of shares in each benchmark (i.e., the netlist size).

In the Verilog implementation of the ISCAS'85 benchmarks, `c7552` comprises more gates than

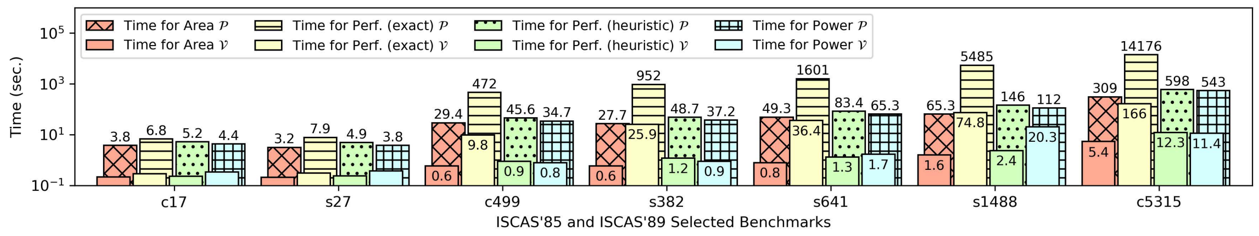


Fig. 9. Timing results for $\mathcal{P}$ and $\mathcal{V}$ of the area, performance (for both exact and heuristic methods), and power modules for selected ISCAS'85 and ISCAS'89 benchmarks. The timings for ISCAS'89 netlists are amortized over 10 cycles. $\mathcal{P}$'s offline costs are reported in Figs. 7 and 8 and are omitted from this plot. Our heuristic method incurs less than 5% error in all cases, and offer significant performance benefits compared to the exact method.

`c5315`; however, after Pythia invokes Yosys to synthesize and transform all the gates with more than two inputs into a series of two-input gates, `c5315` has the most gates compared to the other ISCAS'85 benchmarks, and thus incurs the higher $\mathcal{P}$ and $\mathcal{V}$ costs. We report the number of gates and wires of our synthesized ISCAS benchmarks in Table III. One can observe that the proving time in Pythia scales linearly with the number of shares (i.e., $\lceil \#gates/9 \rceil$) which is quasi-linear ($T \cdot polylog(T)$) in the number of state machine transitions T since each gate involves a fixed number of transitions. Likewise, Pythia's verification time is poly-logarithmic ($polylog(T)$) in T.

Fig. 9 reports our experimental evaluations for both $\mathcal{P}$ and $\mathcal{V}$ using our area, performance, and power estimation modules. As mentioned in Section V-B, the bit-packing optimization cannot be applied to the performance and power estimation modules since they already store multiple bits of information in each state index. As a result, these two modules incur higher overheads than the functional verification module

TABLE III

NUMBER OF GATES AND WIRES GENERATED BY PYTHIA COMPILER AND STATE VECTOR MINIMIZATION AFTER APPLYING GRAPH-COLORING AND BIT-PACKING TECHNIQUES FOR SELECTED BENCHMARKS.

Combinational Benchmark	**c499**	**c880**	**c1355**	**c1908**	**c3540**	**c5315**	**c6288**	**c7552**	**b14_C**	**b15_C**	**b17_C**
# Gates	246	583	1006	1435	2349	3454	2922	2742	4447	7901	24246
# Wires	287	643	1047	1468	2399	3632	2954	2949	4722	8384	25696
Vector Size	48	87	72	189	322	569	63	357	574	620	1791
64-bit Blocks	1	2	2	3	6	9	1	6	9	10	28
Sequential Benchmark	**s349**	**s382**	**s420**	**s444**	**s526**	**s641**	**s838**	**s1488**	**b14**	**b15**	**b17**
# Gates	234	292	322	346	382	455	658	843	4338	8322	25697
# Wires	247	299	314	353	389	494	696	855	4369	8357	25733
Vector Size	57	78	83	82	132	78	170	242	772	1015	3756
64-bit Blocks	1	2	2	2	3	2	3	4	13	16	59

due to the significantly larger state vectors they use. Our performance module using the *exact method* for path delays requires significantly more state indices than any other module to keep track of all wire delays and therefore incurs considerably higher overheads. Nevertheless, *our heuristic method* achieves highly accurate results (less than 5% error from the exact path delays of the ISCAS benchmarks), yet it is several orders of magnitude faster. Likewise, across several ISCAS'85 benchmarks, our heuristic incurs less than 4% error (on average) compared to the results from OpenTimer, a popular open-source static timing analysis tool [54]. Finally, since the area verification module has minimal requirements (i.e., small and constant number of indices for all the netlists), it achieves the fastest performance across all the modules. Moreover, as we observe in the results of Fig. 9, the overhead of our AUX modules scales with the total number of gates in each netlist, while the prover's performance also depends on the number of indices of the state vector.

4) Bit-Packing: Table III summarizes the numbers of gates and wires for our ISCAS benchmarks after Yosys synthesis and demonstrates how our register allocation and bit-packing optimizations drastically reduce the number of intermediate wires. Notably, register allocation is less impactful for very wide netlists, as Pythia maintains all wire values at each circuit level using unique state indices, which attributes to increased overheads. For narrower netlists that have smaller levels (regardless of their depth), our register allocation can significantly reduce state indices and hence Pythia's overheads. For instance, the netlist of `c6288` has 2954 wires, and register allocation can reduce them to just 63, which is a decrease by almost $50\times$ (c.f., row "Vector Size" in Table III). Conversely, netlist `s526`, which achieves a modest decrease of about $3\times$ according to Table III, would benefit less from this optimization. When our bit-packing technique minimizes the state vector size (c.f., row "64-bit Blocks" in Table III) to a value less than 10, the performance benefits are more noticeable.

VII. Related Work

In this Section, we discuss the different categories of zero-knowledge proof systems and our choice of zk-STARK as the back-end cryptographic protocol of Pythia. The first category involves systems that need a *trusted setup phase for each different computation* they want to prove. This line of work started from the construction proposed by Gennaro et al. [25] using quadratic arithmetic programs and continued with the works in [24], [27], [28]. More recent proof systems utilize *universal and updatable trusted setups* that are based on common reference strings (e.g., [55]). Their advantage compared to the previous category is that they do not require a trusted pre-processing phase for each circuit, but only a single setup for all circuits. Although this method requires only one trusted setup, neither this nor the first category are applicable for our threat model since a malicious party that gains access to the trusted setup phase can always forge false proofs.

The last category includes systems with a *transparent setup* (i.e., no requirement for a trusted third-party to initialize the system). This includes preliminary works in [56]–[58], as well as zk-STARK-based systems [18], [19]. Pythia's back-end relies on zk-STARK as it supports universal, trustless setup in line with our threat model; moreover, verifiers in zk-STARK protocols can achieve the best-in-class performance, while the low-overhead proving times [18].

The authors of [15] proposed a novel technique that leverages homomorphic encryption (a cryptographic primitive that allows performing meaningful operations on encrypted data) to enable secure outsourcing and evaluation of 3PIP designs using encrypted input vectors. In a similar direction, the authors of [17] propose a framework that converts Verilog HDL programs into homomorphic circuits with equivalent functionality, and evaluates them using encrypted input vectors. Both of these approaches allow untrusted third parties to evaluate an IP while preserving the privacy of the input test vectors. Nevertheless, since homomorphic operations only protect data privacy but not applied function (i.e., Boolean gates remain unencrypted in the source file), the netlist designs are not protected as in Pythia.

Previous work has also focused on proving the correctness and various properties of 3PIP designs using formal logic verification [12], [59]. These techniques mostly focus on formally checking the correctness of a netlist or a cryptographic protocol to ensure the design is free of malicious logic, such as hardware Trojans (e.g., [60]). Yet, such methods focus on comparing the netlist with a formal mathematical model of the circuit and exclude any privacy protections for the IP. Conversely, the threat model of this work (Section II-D) assumes that cheating verifiers have incentives to extract information about the IPs before a financial transaction is completed. In this context, Pythia offers a novel privacy-preserving approach to such deadlocks, by enabling system integrators to verify that 3PIP vendors possess an IP with certain functional, performance, area, and power constraints.

Recent research directions have also focused on obfuscation methods that aim to prevent IP theft of circuit designs, by inserting additional gates into the circuit in order to hide its implementation [6]. Likewise, watermarking [61] and fingerprinting [62] methods embed author signatures in the design to deter IP theft, as they allow tracking the source of leakage in case of piracy violations. While such techniques mitigate the risk of IP theft, they come with important limitations: First, these techniques only make it harder for the attackers but do not guarantee that 3PIPs can not be leaked. Furthermore, all the aforementioned techniques tamper with the IP design and, more importantly, they assume that verification takes place *after the IP is outsourced* to the system integrator. In contrast, Pythia is designed for privacy and makes it impossible for such designs to be leaked in the first place, as system integrators never access the IP while verifying its properties. Notably, in this work, we can prove both the functionality of a 3PIP, as well as estimate important properties (area, performance, and power) *without altering the netlists* in any way. To the best of our knowledge, Pythia is the first framework that employs ZKPs in the context of privacy-preserving IP verification, which is an exciting research direction for integrated circuits.

VIII. Concluding Remarks

In this paper, we have proposed the first-of-its-kind Pythia framework for privacy-preserving IP verification. Pythia features a custom compiler that translates any circuit into a specialized encoding that is then evaluated in zero-knowledge using public test vectors; our method generates cryptographic proofs to attest the faithful evaluation of test inputs on a netlist, yet the netlist itself remains a black box for the verifier. Security is guaranteed by Pythia's back-end that leverages the provably secure zk-STARK protocol. Our methodology also supports different auxiliary modules that can further estimate the area, performance, and power consumption of a netlist in zero-knowledge. Moreover, Pythia leverages authenticated hash digests to prove that the secret netlist was not altered across different test vectors. Our methodology is complemented by optimization techniques that reduce performance overheads and memory requirements using register allocation and bit-packing. Notably, Pythia can automatically split a given netlist into multiple shares that can be evaluated in parallel. Finally, we have demonstrated Pythia's versatility using combinational and sequential circuits and have investigated the impact of the netlist size on Pythia's performance.

References

[1] J. Gubbi *et al.*, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] K. Mandula *et al.*, "Mobile based home automation using Internet of Things (IoT)," in *ICCICCT*. IEEE, 2015, pp. 340–343.

[3] M. Rostami *et al.*, "Hardware security: Threat models and metrics," in *ICCAD*. IEEE/ACM, 2013, pp. 819–823.

[4] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[5] M. Tehranipoor and C. Wang, *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[6] J. A. Roy, F. Koushanfar, and I. L. Markov, "EPIC: Ending piracy of integrated circuits," in *DATE*. ACM, 2008, pp. 1069–1074.

[7] R. Torrance and D. James, "The state-of-the-art in IC reverse engineering," in *CHES*. Springer, 2009, pp. 363–381.

[8] E. Castillo *et al.*, "IPP@HDL: Efficient Intellectual Property Protection Scheme for IP Cores," *IEEE TVLSI*, vol. 15, no. 5, pp. 578–591, 2007.

[9] P. Chauhan *et al.*, "Verifying IP-core based system-on-chip designs," in *ASIC/SOC*. IEEE, 1999, pp. 27–31.

[10] A. Deshpande, "Verification of IP-Core based SoC's," in *ISQED*. IEEE, 2008, pp. 433–436.

[11] 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.

[12] 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.

[13] B. Keng and A. Veneris, “Path-Directed Abstraction and Refinement for SAT-Based Design Debugging,” *IEEE TCAD*, vol. 32, no. 10, pp. 1609–1622, 2013.

[14] G. Moretti *et al.*, “Your Core – My Problem? Integration and Verification of IP,” in *DAC*. IEEE/ACM, 2001, pp. 170–171.

[15] C. Konstantinou, A. Keliris, and M. Maniatakos, “Privacy-preserving functional IP verification utilizing fully homomorphic encryption,” in *DATE*. EDAA, 2015, pp. 333–338.

[16] N. G. Tsoutsos and M. Maniatakos, “The HEROIC framework: Encrypted computation without shared keys,” *IEEE TCAD*, vol. 34, no. 6, pp. 875–888, 2015.

[17] 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.

[18] E. Ben-Sasson *et al.*, “Scalable zero knowledge with no trusted setup,” in *CRYPTO*. Springer, 2019, pp. 701–732.

[19] D. Mouris and N. G. Tsoutsos, “Zilch: A framework for deploying transparent zero-knowledge proofs,” *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3269–3284, 2021.

[20] S. Goldwasser, S. Micali, and C. Rackoff, “The Knowledge Complexity of Interactive Proof-Systems,” *Journal on computing*, vol. 18, no. 1, pp. 186–208, 1989.

[21] B. Adida, “Helios: Web-based Open-Audit Voting.” in *USENIX Security*, vol. 17, 2008, pp. 335–348.

[22] A. Kosba *et al.*, “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts,” in *S&P*. IEEE, 2016, pp. 839–858.

[23] O. Goldreich and A. Kahan, “How to construct constant-round zero-knowledge proof systems for NP,” *Journal of Cryptology*, vol. 9, no. 3, pp. 167–189, 1996.

[24] B. Parno *et al.*, “Pinocchio: Nearly practical verifiable computation,” in *S&P*. IEEE, 2013, pp. 238–252.

[25] R. Gennaro *et al.*, “Quadratic span programs and succinct NIZKs without PCPs,” in *EUROCRYPT*. Springer, 2013, pp. 626–645.

[26] E. Ben-Sasson *et al.*, “Fast reductions from RAMs to delegatable succinct constraint satisfaction problems,” in *ITCS*. ACM, 2013, pp. 401–414.

[27] ——, “SNARKs for C: Verifying program executions succinctly and in zero knowledge,” in *CRYPTO*. Springer, 2013, pp. 90–108.

[28] ——, “Succinct non-interactive zero knowledge for a von Neumann architecture,” in *USENIX Security*, 2014, pp. 781–796.

[29] Y. Zhang *et al.*, “vRAM: Faster verifiable RAM with program-independent preprocessing,” in *S&P*. IEEE, 2018, pp. 908–925.

[30] M. Bellare and O. Goldreich, “On Defining Proofs of Knowledge,” in *CRYPTO*. Springer, 1992, pp. 390–420.

[31] J. Groth, “On the size of pairing-based non-interactive arguments,” in *EUROCRYPT*. Springer, 2016, pp. 305–326.

[32] E. Ben-Sasson *et al.*, “Fast Reed-Solomon interactive oracle proofs of proximity,” in *ICALP*. EATCS, 2018.

[33] A. Shamir, R. L. Rivest, and L. M. Adleman, “Mental poker,” in *The mathematical gardner*. Springer, 1981, pp. 37–43.

[34] M. Naor, “Bit commitment using pseudorandomness,” *Journal of Cryptology*, vol. 4, no. 2, pp. 151–158, 1991.

[35] M. Luby and C. Rackoff, “How to construct pseudo-random permutations from pseudo-random functions,” in *CRYPTO*, vol. 85, 1985, p. 447.

[36] R. Beaulieu *et al.*, “The SIMON and SPECK lightweight block ciphers,” in *DAC*. ACM/EDAC/IEEE, 2015, pp. 1–6.

[37] V. Shoup, “Sequences of games: a tool for taming complexity in security proofs.” Cryptology ePrint Archive, Report 2004/332, 2004.

[38] J. L. Carter and M. N. Wegman, “Universal classes of hash functions,” *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[39] N. G. Tsoutsos and M. Maniatakos, “Efficient Detection for Malicious and Random Errors in Additive Encrypted Computation,” *IEEE Transactions on Computers*, vol. 67, no. 1, pp. 16–31, 2017.

[40] T. Krovetz, “Message authentication on 64-bit architectures,” in *SAC*. Springer, 2006, pp. 327–341.

[41] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2014.

[42] C. Wolf, J. Glaser, and J. Kepler, “Yosys - A Free Verilog Synthesis Suite,” in *Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[43] F. J. Kurdahi and A. C. Parker, “PLEST: a program for area estimation of VLSI integrated circuits,” in *DAC*. ACM/EDAC/IEEE, 1986, pp. 467–473.

[44] M. Nemani and F. N. Najm, “High-level area and power estimation for VLSI circuits,” *IEEE TCAD*, vol. 18, no. 6, pp. 697–713, 1999.

[45] R. F. Sproull and I. E. Sutherland, “Logical effort: Designing for speed on the back of an envelope,” *IEEE Advanced Research in VLSI*, vol. 9, p. 219, 1991.

[46] P. A. Beerel and T. H.-Y. Meng, “Automatic gate-level synthesis of speed-independent circuits,” in *ICCAD*. IEEE/ACM, 1992, pp. 581–586.

[47] I. Ciofi *et al.*, “Impact of Wire Geometry on Interconnect RC and Circuit Delay,” *IEEE Transactions on Electron Devices*, vol. 63, no. 6, pp. 2488–2496, 2016.

[48] A. Ghosh *et al.*, “Estimation of average switching activity in combinational and sequential circuits,” in *DAC*, vol. 29. ACM/EDAC/IEEE, 1992, pp. 253–269.

[49] J. Monteiro *et al.*, “Estimation of average switching activity in combinational logic circuits using symbolic simulation,” *IEEE TCAD*, vol. 16, no. 1, pp. 121–127, 1997.

[50] N. H. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson Education India, 2015.

[51] F. Brglez, “A neural netlist of 10 combinational benchmark circuits,” *IEEE ISCAS: Special Session on ATPG and Fault Simulation*, pp. 151–158, 1985.

[52] F. Brglez, D. Bryan, and K. Kozminski, “Combinational profiles of sequential benchmark circuits,” in *ISCAS*. IEEE, 1989, pp. 1929–1934.

[53] F. Corno, M. Reorda, and G. Squillero, “RT-level ITC’99 benchmarks and first ATPG results,” *IEEE Design Test of Computers*, vol. 17, no. 3, pp. 44–53, 2000.

[54] T.-W. Huang and M. D. Wong, “OpenTimer: A high-performance timing analysis tool,” in *ICCAD*. IEEE, 2015, pp. 895–902.

[55] A. Chiesa *et al.*, “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS,” in *Eurocrypt*. Springer, 2020, pp. 738–768.

[56] J. Bootle *et al.*, “Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting,” in *Eurocrypt*. Springer, 2016, pp. 327–357.

[57] R. S. Wahby *et al.*, “Doubly-efficient zkSNARKs without trusted setup,” in *S&P*. IEEE, 2018, pp. 926–943.

[58] B. Bünz *et al.*, “Bulletproofs: Short proofs for confidential transactions and more,” in *S&P*. IEEE, 2018, pp. 315–334.

[59] C. Kern and M. R. Greenstreet, “Formal verification in hardware design: a survey,” *ACM TODAES*, vol. 4, no. 2, pp. 123–193, 1999.

[60] N. G. Tsoutsos, C. Konstantinou, and M. Maniatakos, “Advanced techniques for designing stealthy hardware Trojans,” in *DAC*. IEEE/ACM, 2014, pp. 1–4.

[61] A. B. Kahng *et al.*, “Watermarking techniques for intellectual property protection,” in *DAC*. IEEE/ACM, 1998, pp. 776–781.

[62] A. E. Caldwell *et al.*, “Effective iterative techniques for fingerprinting design IP,” *IEEE TCAD*, vol. 23, no. 2, pp. 208–215, 2004.