Pythia: Intellectual Property Verification in Zero-Knowledge

Dimitris Mouris and Nektarios Georgios Tsoutsos
Electrical and Computer Engineering, University of Delaware
E-mail: {jimouris, tsoutsos}@udel.edu

***Abstract*—The contemporary IC supply chain depends heavily on third-party intellectual property (3PIP) that is integrated to in-house designs. As the correctness of such 3PIPs should be verified before integration, one important challenge for 3PIP vendors is proving the functionality of their designs while protecting the privacy of circuit implementations. In this work, we present Pythia that employs zero-knowledge proofs to enable vendors convince integrators about the functionality of a circuit without disclosing its netlist. Pythia automatically encodes netlists into zero knowledge-friendly format, evaluates them on different inputs, and proves correctness of outputs. We evaluate Pythia using the ISCAS'85 benchmark suite.**

***Index Terms*—Hardware security, trustworthy hardware, intellectual property verification, zero-knowledge proofs, IP theft**

I. Introduction

In this new interconnected era of the Internet of Things (IoT), System-on-Chip (SoC) have conquered the market due to less power and area consumption, increased reliability and functionality compared to multi-chip designs. IoT enables an extensive set of applications such as transportation systems, healthcare, home automation, and many more, the majority of which utilize SoCs. SoCs combine the required electronic circuits of various computer components onto a single, integrated chip (IC). The contemporary IC supply chain, involves designing some components in-house, procuring a variety of Intellectual Property (IP) cores from third-party foundries and integrating them together to generate the IC [1].

As the demand of IC rapidly grows, the number of third party IP (3PIP) vendors is increasing as well. This increase has attracted a variety of untrusted IP vendors and malicious users trying to trick honest parties for financial gain [2]. At the same time, hardware 3PIP vendors make their IPs reusable and provide design standards and guidelines so they can be utilized by multiple design layouts to increase profit [3]. However, focusing more on IP functionality and performance than security, renders IP piracy a significant threat since system level analysis [4] and reverse engineering [5] become easier.

IP core verification is a crucial component of SoC design in the modern IC business model. 3PIP vendors design circuits with respect to some agreed-upon functional specifications, provided by system integrators (i.e., IP consumers), and try to prove the functionality of their designs while protecting the privacy of circuit implementations. Achieving sufficient verification and high testability are the major bottlenecks in the IC supply chain; i.e., to confirm that the circuit complies to the specified properties. Many related solutions have been proposed such as application-specific instruction-set processors [6], SAT solvers [7], formal logic verification [8], simulation [9] and homomorphic encryption [10] based methods.

In this paper, we utilize zero-knowledge proofs to address the trust issues between 3PIP vendors and system integrators. We propose the Pythia framework that enables IP consumers verify that a *potentially untrusted* vendor possesses an IP that satisfies some agreed-upon properties *without having access to it* (i.e., gaining zero knowledge about the IP). Pythia automatically encodes a netlist into a zero knowledge-friendly format, evaluates it given test input vectors and enables proving correctness of the output. Having as our back-end the `libSTARK` library [11] that can be used to argue about the integrity of a computation (i.e., creating proof that a computation was executed correctly), Pythia implements a novel circuit simulator as a zero-knowledge state machine. Moreover, Pythia implements a special encoding to convert input test vectors to a format compatible with our circuit simulator, which allows evaluation of the Boolean circuit operations defined in the 3PIP netlist to be carried out in zero knowledge, without revealing the netlist itself. Specifically, the 3PIP netlist is treated as a private (secret) input to Pythia's circuit simulator, while the test vectors are treated as public inputs; Pythia creates cryptographic proofs that the circuit simulator has faithfully evaluated the secret netlist on the public inputs and that the return output is correct. In this work, our contributions can be summarized as follows:

- **Simulator:** Development of a novel circuit simulator as a zero-knowledge state machine that can evaluate Boolean circuits on any input test vector, without ever revealing the netlist.
- **Compiler:** Automatic compilation of Boolean circuit netlists into a serialized, zero knowledge-friendly format that eliminates inter-dependencies between intermediate gate inputs and upstream gate outputs.
- **Optimizer:** Internal state minimization for Pythia's circuit simulator leveraging bit-packing and graph-coloring techniques for optimal allocation of intermediate wire values to the state vector.
- **Parallelism:** Pythia automatically divides our compiled 3PIP netlists into independent shares that can be simulated *in parallel*, while cryptographically proving continuity between all consecutive shares.

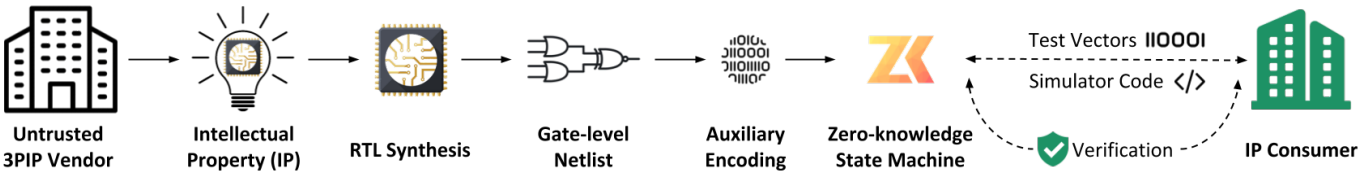

Fig. 1. **Overview of Pythia:** (a) The 3PIP vendor (prover) possesses an IP described in a Hardware Description Language. (b) The prover synthesizes the IP and generates a gate-level netlist. (c) The 3PIP vendor determines the evaluation order of the gates and minimizes the number of intermediate wire values required to evaluate the netlist. (d) The prover transforms the IP into zero-knowledge friendly encoding for the state machine simulator. (e) The system integrator (verifier) supplies a set of public inputs. (f) The 3PIP vendor evaluates the IP circuit and proves in zero-knowledge the computational integrity of the evaluation through interaction with the verifier.

Roadmap: In Section II we offer a brief discussion on zero-knowledge proofs, universal hashing, and our threat model, while in Section III we present our zero-knowledge IP verification methodology. Finally, Section IV discusses our evaluations, followed by related work and our conclusions in Sections V and VI, respectively.

II. Preliminaries

A. Basics of Zero-Knowledge Proofs

A zero-knowledge proof is a cryptographic protocol between two parties; a *prover* and a *verifier* [12]. The prover's goals is to convince the verifier that she *knows* a correct secret input w (known as a *witness*) to a publicly-known program $\mathcal{P}$, so that the output of $\mathcal{P}$ on secret w and some additional public input x is a fixed value $y = \mathcal{P}(x, w)$. As a simple example, $\mathcal{P}$ can be a program that applies a hashing algorithm (e.g., SHA-256) on w and compares the resulting hash with x to return true or false as follows:

$$\mathcal{P}(x, w) := \text{if } \mathcal{H}(w) = x \text{ return } True \text{ else return } False.$$

To convince the verifier, the prover executes $\mathcal{P}(x, w)$ locally using inputs x, w and records the execution transcript as a sequence of intermediate states on a special state machine implementing $\mathcal{P}$ [13]. Since the transcript corresponds to the execution of publicly-known program $\mathcal{P}$, all state transitions in the execution transcript must satisfy certain cryptographic constraints that are expressed using low-degree polynomials, which are eventually checked by the verifier [11]. By design, the zero-knowledge protocol does not leak any information about w beyond that fact that the verifier has faithfully executed $\mathcal{P}(x, w)$ and the output is y (i.e., $True$ or $False$). Moreover, the protocol can guarantee that the verifier is able to detect if the prover is cheating by falsely claiming knowledge of the witness input (i.e., the prover cannot convince the verifier if she does not actually know the correct w) [14].

Pythia generalizes the example above using a full circuit simulator as the public program $\mathcal{P}$: our simulator takes a secret 3PIP netlist as witness input w, together with a public input test vector x, and evaluates that netlist on the test input. The simulation output is a result vector y.

B. Lightweight Pseudorandom Functions with Extended Input

A pseudorandom function (PRF) $\mathcal{F}$ is a deterministic algorithm that combines a secret key S with an input block X and returns an output block Y that is computationally indistinguishable from truly random bits [15]. In practice, one common class of PRFs includes secure block ciphers, such as Speck [16], which use a secret key to transform a plaintext block into a ciphertext block. Even though PRFs have fixed input block sizes, it is possible to extend their input length to arbitrary sizes using a universal hash function (UHF). Indeed, the $PRF(UHF(\cdot))$ construction (i.e., universal hash and then encrypt [17]) is also a secure PRF [18, Section 4].

Based on the Carter and Wegman blueprint, a UHF $\mathcal{U}$ defined over ℓ input blocks (m_1 to m_ℓ) and a secret key k can be constructed as a polynomial of degree-ℓ modulo a prime number q that is evaluated at point k. In particular, $\mathcal{U}(k, m_1, \ldots, m_\ell) = \sum_1^\ell m_i k^i \bmod q$, with $m_i \in \mathbb{Z}_q$, is a lightweight UHF with a collision probability $\varepsilon \leq \ell/q$ [19]. Notably, $\mathcal{U}$ can be computed iteratively using Horner's method, and can be encrypted using the Speck block cipher to construct a secure PRF with input extended over ℓ blocks.

C. Threat Model

In this work, we tackle the problem of mutual mistrust between 3PIP vendors and system integrators using zero-knowledge proofs. More specifically, our threat model involves a vendor claiming that she possesses an IP, and an integrator interested in buying that IP. Our model assumes a cheating vendor that may have an incentive to deceive the integrator and attempt to sell an IP that does not meet the agreed-upon functionality; in this case, the buyer needs to test the IP using multiple input vectors and check the correctness of the outputs. Likewise, we assume a cheating buyer that may attempt to obtain the IP before paying, so the vendor cannot disclose the corresponding netlist until after payment is committed.

III. The Pythia Framework

A. Overview of our framework

We present *Pythia*, a framework for privacy-preserving 3PIP verification. Pythia enables vendors to convince system integrators that they possess 3PIPs that meet some agreed-upon functional specifications without revealing their designs (i.e., in zero-knowledge). Fig. 1 demonstrates the Pythia framework, which is outlined by the following steps:

1) The 3PIP vendor (prover) synthesizes the circuit described in a Hardware Description Language (HDL), such as Verilog, and creates a gate-level netlist.
2) Consecutively, the prover determines the evaluation order of the gates and minimizes the number of intermediate

wires required to evaluate the circuit leveraging Pythia's bit-packing and graph coloring techniques.

3) The 3PIP vendor transforms the IP into a zero knowledge-friendly format that can be used by Pythia's state machine as witness input.
4) The IP user (verifier) provides a test vector for the netlist that is supplied as public input to Pythia's state machine.
5) The 3PIP vendor evaluates the private netlist with the public test vector and computes a public output.
6) The two parties interact using Pythia and the 3PIP vendor proves in zero-knowledge the correctness of the computation.

The following sections elaborate more on the various components of our Pythia framework.

Pythia's back-end: Pythia employs `libSTARK` [11] to construct a custom state machine that evaluates logic circuits. In more details, the programming interface of libSTARK enables the development of assertions about computational integrity, which are used to prove the correctness of the execution trace and transitions of arbitrary state machines. The initial state of Pythia's machine is a vector of zeros and after each step of the computation a new state vector is appended to the execution trace; the latter can be envisioned as a table comprising a sequence of state vectors that represent the computation. The state machine can modify its state based on a broad range of operations, including arithmetic and bitwise operations as well as conditional decisions.Using its back-end, Pythia imposes polynomial constraints to the execution trace and proves to the verifier their satisfiability (e.g., generates a proof of correct evaluation of the gate, while keeping the actual gate a secret).

B. From IP Netlists to Zero-Knowledge Simulation

IP Core Transformation: In our approach, we assume 3PIPs are netlists described in the Electronic Design Interchange Format (EDIF). Pythia enables automatic compilation of EDIF netlists into a zero knowledge-friendly format to transform the IP logic into a form that can be interpreted and utilized by its back end. Pythia's compiler parses the netlist and eliminates inter-dependencies between intermediate gate inputs and upstream gate outputs by creating a directed acyclic graph (DAG). Using the DAG, Pythia determines the evaluation order of the circuit's gates by running a topological sort algorithm and resolves all the dependencies in the netlist. Consecutively, our compiler transforms any gates of the circuit that take more than two inputs into a sequence of two-input 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 can be used as witness input in our zero-knowledge circuit simulator.

Circuit Simulator: The core of the Pythia framework is the development of a circuit simulator as a zero-knowledge state machine that can evaluate Boolean circuits on any input test vector, without revealing the netlist. The state machine reads a series of logic gates from the private input along with a number of public binary inputs and evaluates their output. For each gate that Pythia consumes from the private input, it reads two public inputs and depending on the gate type, the state machine determines what operation to perform on the two input values, computes the result and proves the integrity of that computation. However, since many gates rely on the outputs of preceding gates, after Pythia evaluates all the gates that take constant binary inputs, the inputs for intermediate gates have to be supplied from the outputs of the already evaluated gates. As Pythia stores these outputs at predetermined indexes of the state vector, the private input should encode the index information as well.

Algorithm 1 State Machine for Circuit Simulation

Input: Public inputs, Private inputs

1: **procedure** EVALCIRCUIT
2: **for** each public input **do** Initialize $StateVector$
3: $H \leftarrow 0$ ▷ Keeps track of the PRF of the IP
4: **for** each private input **do**
5: Read $gateID$, in_0, in_1, out
6: $H \leftarrow PRF(H, gateID, in_0, in_1, out)$
7: **if** $gateID = AND$ **then** $out \leftarrow in_0$ & in_1
8: **else if** $gateID = OR$ **then** $out \leftarrow in_0 \mid in_1$
9: **else if** $gateID = \ldots$ **then** $\ldots$
10: $StateVector \leftarrow Update(StateVector, out)$
11: **return** $StateVector$, H

More specifically, after each gate identifier in the private input, Pythia encodes three state vector indexes, two for the gate input values and one for the output of the gate. Recall that both the prover and the verifier have access to the test-vectors and the state machine simulator, but only the former knows the gates and the indexes that are being used to evaluate them. With Pythia, the verifier never see the 3PIP netlist, yet she can be convinced that the prover correctly executed the simulator code with the public input-vector and generated the public output.

To enable 3PIP verification, the system integrator has to supply a variety of different input-vectors to ensure that the 3PIP satisfies the agreed-upon functionality. However, since during zero-knowledge verification the system integrator does not acquire any knowledge about the encoded 3PIP (private input) used to generate the public output, she also needs to be convinced the same 3PIP was used across different executions using different test vectors. To address this requirement while protecting the confidentiality of the 3PIP, Pythia employs a PRF that generates authenticated digests from the 3PIP netlist during evaluation. The choice of PRF is crucial, considering the high overhead the prover will incur due to the increased number of instructions to compute the PRF along with the circuit simulation. Thus, we employ the lightweight PRF with extended input discussed in Section II-B.

The functionality of Pythia's circuit simulator is summarized in Alg. 1. First, the state machine is initialized with the verifier's chosen public inputs (line 2). Then, our simulator reads the gate identifier and three indexes (i.e., two inputs and one output) and computes an integrity measurement using the PRF (line 6). The simulator determines what operation to perform on the inputs, evaluates the output, updates the state

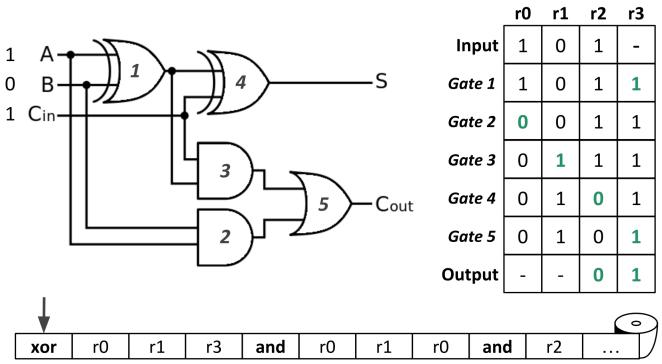

Fig. 2. **Adder evaluation:** The numbers on the gates denote the evaluation order, also depicted in the first column of the table (execution trace) on the right-hand side. The green values represent which variable changed after the evaluation of the gate denoted by the row number. The variables $r0$–$r3$ can be interpreted as 4 separate or as one 4-bit block. The private tape that encodes a part of the above circuit is depicted on the bottom.

vector and repeats the process for the next gate. Ultimately, the simulator outputs the final state vector and the computed PRF digest of the IP.

C. Pythia's Optimizer

Assigning each intermediate wire to a different state-vector index could blow up the state size required to verify even a small circuit. Thus, Pythia adopts register allocation principles using graph coloring from the compilers literature [20] to utilize a smaller number of indexes in its state vector more efficiently. During the IP transformation phase, the 3PIP vendor performs index allocation in the state vector for the input, output and all the intermediate wires of the netlist, which significantly reduces the total number of required indexes.

In Fig. 2, we demonstrate how register allocation techniques can reduce the number of indexes for an adder evaluation. Initially, the circuit would require 8 total wires; 3 input, 2 output as well as 3 more for the intermediate results of gates 1, 2 and 3. Using our index allocation approach, we can evaluate this circuit with just 4 wires, as shown in the execution trace in the right-hand side of Fig. 2. The first row corresponds to the initialization of the state vector using the public input, and each subsequent row corresponds to the evaluation of the next gate (in each row, we highlight the output of a gate in green). The bottom of the figure shows a part of the transformed adder circuit (i.e., the 3PIP witness); initially, the state machine reads the XOR operation along with the two input variables (`r0`, `r1`), evaluates the result, stores it to the output variable (`r3`), before continuing to the next operation.

Bit-Packing: As IP circuits grow larger, the number of wires holding intermediate values can increase exponentially, even for state machines that apply register allocation techniques. Even though Pythia's state vector does not have a size limit, the more indexes in the state vector, the longer the proving time. To address this concern, and given the fact that we only store binary values in the wires, Pythia implements a bit-packing optimization that first organizes wires into blocks, and each index of the state vector holds one block (the individual bits within each block have a separate sub-index). Specifically, our bit-packing scheme utilizes 64-bit blocks that hold 64 intermediate binary values at different sub-indexes within the block, which significantly reduces the number of indexes in the state vector. Notably, to enable this optimization, the witness input should encode both state vector indexes and block sub-indexes. Although Pythia's state machine now uses slightly more state transitions to read and write in the correct sub-indexes of each block (e.g., shift operations to isolate the correct bits), the overhead of utilizing bit-packing is negligible compared to the cost of having to verify state vectors with larger index sizes in the execution trace. With bit-packing, the adder in Fig. 2 that normally requires a state vector holding at least 4 variables, can now use a state vector with a single 4-bit block.

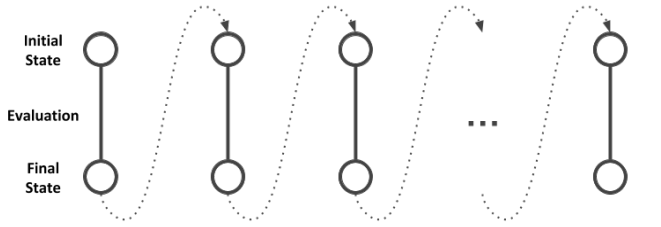

Fig. 3. Chaining execution to enable parallel verification. Pythia divides large executions into multiple shares and pre-computes the intermediate states locally. The simulator computes the PRF digest of the machine state and compares it with the digest provided in the public tape to verify its integrity. Each share can be verified independently and in parallel.

D. Execution Parallelism

The execution trace of each IP simulation has an initial state of zeros and a final state that denotes the values on the output wires of the netlist that was evaluated. Given an execution trace of a netlist evaluation, we can break down the problem proving the faithful execution of the whole trace into the problem of proving the execution of two – or more – smaller traces (dubbed *shares*) while also verifying a valid transition between them to create a valid *execution chain*. This is illustrated in Fig. 3. To convince the IP consumer that the shares are decompositions of the original transcript, while preserving the confidentiality of the intermediate state vectors of Pythia's simulator, we employ the lightweight PRF discussed earlier to compute integrity measurements of the state vectors. The prover shares these authenticated digests with the verifier, but only the former knows the actual vectors that produced that digest (the verifier selects Speck encryption key). Finally, the system integrator can verify that the PRF digest at the end of each share is the same that gets extended at the beginning of the next share.

Even though we have broken down a long execution trace of a netlist evaluation into shares that can be chained together to compute the same result, we cannot directly exploit parallelism since each share depends on the output (i.e., machine state) of the previous one. To address this problem, Pythia allows the 3PIP vendor to simulate the netlist locally (i.e., without creating a proof), pre-calculate the correct state vector digests and use them to initialize each share. Notably, the local execution overhead is negligible compared to the actual proving

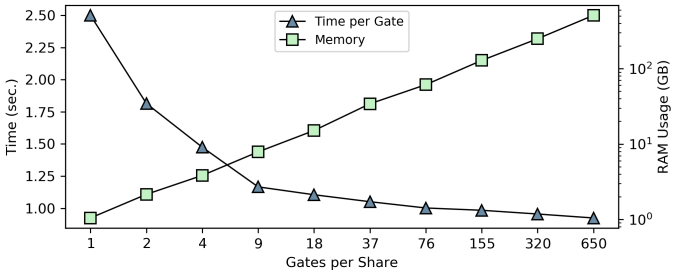

Fig. 4. Memory/Efficiency per gate trade-off. Horizontal axis shows the maximum number of gates per share that can be simulated in less than a power of 2 state machine transitions.

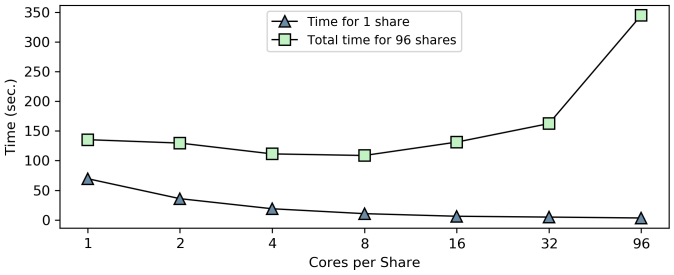

Fig. 5. Time measurements for verifying 1 and 96 shares with different number of parallel threads per share.

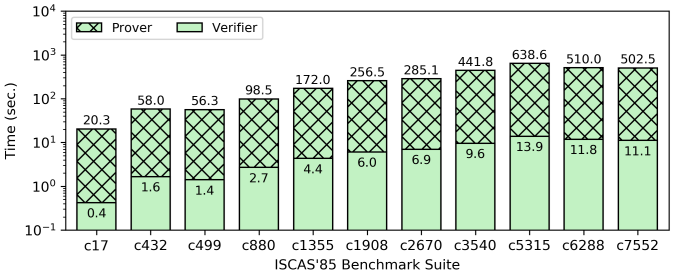

Fig. 6. Pythia's experimental results for the ISCAS-85 benchmark suite.

time. Consecutively, the zero-knowledge IP verification can take place in parallel since there are no more dependencies between the shares; for soundness, the IP consumer needs to verify that the actual digests computed in each share are the ones initialized in the public tape of the next share.

IV. Experimental Results

A. Experimental Setup

In this section, we evaluate Pythia using the ISCAS'85 benchmark suite. The benchmarks have been synthesized to produce EDIF netlists using the Yosys Open SYnthesis Suite for Verilog RTL synthesis [21]. Experimental results are obtained on a m5.24xlarge AWS EC2 instance running with two Intel Xeon Platinum 8175M processors at 2.5 GHz and hyper-threading for a total of 96 virtual cores and 748 GB RAM. The system is running Ubuntu 18.04 with the 4.15.0 Linux kernel, g++ 7.4.0 compiler and Python 3.6.8.

B. Performance Evaluation

Pythia's back-end incurs different time and memory overheads depending the size of the execution trace. As the execution traces become bigger, both the execution time and the memory required for the prover are increasing. However, any two traces with size less than a power of 2 take roughly the same proving time and same memory. For example, the time and memory overheads to prove a trace with 942 steps are approximately the same as the ones for a trace with 1020 steps (since both are less than 2^{10}).

Splitting Shares Trade-off: In Fig. 4 we present the proving time per gate for varying number of gates per share (left vertical axis) as well as the memory usage for a variety of gates per share (right vertical axis). Although the total proving time is increasing as the execution traces become larger, the more gates we verify in a single trace, the less the proving time per gate. This result indicates that the fastest prover timings can be achieved by creating as big shares as possible. However, since the required memory (green squares in Fig. 4) scales linearly as the execution trace increases, it becomes impractical to verify more than 650 gates per share. Thus, the fastest proving time is highly dependable to the number of shares that we are able to prove in parallel.

Optimal Parallelization: Taking into consideration the results of Fig. 4, we study how we can exploit parallelism. In order to verify 96 shares in parallel, each share cannot contain more than 9 gates, as the memory required by the prover would increase to impractical levels. The blue triangles in Fig. 5 show the proving time for 1 share that contains 9 gates with different number of cores. Doubling the number of processors almost halves the execution time, but, as we reach to 96 cores, the trend follows Amdahl's law and the ideal 2x speedup starts diminishing. This is an indicator that we should not allocate too many cores on a single execution and that we should distribute the cores to verify more shares in parallel. The green squares in Fig. 5 show how our protocol scales for 96 shares. On one extreme, using 1 core per share we can verify all 96 shares in parallel, while on the other extreme we can use all 96 cores to verify 1 share at a time and repeat this 96 times. As we observe from our experiments, using 8 cores per share and verifying 12 shares in parallel 8 times (to verify the total 96 shares) achieves the fastest proving time.

Experimental Timings: We evaluate Pythia using the ISCAS'85 benchmark suite with random input test-vectors. Although different input patterns trigger different gates to be simulated, the total cost for each benchmark is approximately the same. Fig. 6 illustrates our performance evaluations for a series of ISCAS benchmarks divided to 9 gates per share and verifying each share with 8 parallel cores. Both prover and verifier timings are highly dependent on the number of execution steps, and because each share consists of 9 exactly gates, the time to prove one share is constant. Thus, the primary factor that affects performance is the total number of shares in each benchmark. Notably, after Pythia transforms all gates with more than two inputs into a series of gates with two inputs, `c5315` has the most gates compared to the other ISCAS'85 benchmarks, and thus the higher prover and verifier timings. As we observe, verification time in Pythia is poly-logarithmic ($polylog(T)$) to the number of execution steps T, while prover time is quasi-linear in T ($T \cdot polylog(T)$). Finally, the time for the local prover phase to generate the PRF digests

TABLE I
STATE VECTOR MINIMIZATION AFTER APPLYING GRAPH-COLORING AND BIT-PACKING TECHNIQUES

Benchmark	c17	c432	c499	c880	c1355	c1908	c2670	c3540	c5315	c6288	c7552
Wires	17	350	287	643	1047	1468	1899	2399	3632	2954	2949
Vector Size	5	39	48	87	72	189	242	322	569	63	357
64-bit Blocks	1	1	1	2	2	3	4	6	9	1	6

and enable parallel verification is negligible and is not shown in Fig. 6.

Bit-Packing: Table I summarizes how Pythia's register allocation and bit-packing techniques reduce the required number of intermediate wires for the ISCAS'85 benchmarks. Recall that all wires (input, intermediate and output) require unique indexes in the state vector to store their values. Some of the benchmarks have over one thousand wires, rendering it computationally expensive to hold them all in different offsets at the same time. Using graph coloring for register allocation, we are able to significantly reduce the required number of wires (Table I, third row), yet some of the benchmarks still incur increased overheads. By employing our bit-packing scheme, however, we are able to evaluate all ISCAS'85 benchmarks with no more than 9 64-bit blocks (Table I, last row).

V. Related Work

In [10], Konstantinou et al. proposed a transformation of 3PIP designs that leverages homomorphic encryption and employs encrypted input vectors. Although this approach allows third parties to perform IP verification without having access to the unencrypted vectors, the netlist designs can be leaked since homomorphic operations does not hide the type of operation (only the data). Other methods leverage formal logic verification [8], [22] to prove to system integrators that IPs satisfy agreed-upon properties and are not subject to hardware attacks, such as hardware Trojans [23]. These methods, however, focus on efficient verification by the system integrators and do not consider that the verifiers may have incentives to obtain the IP before paying, thus do not try to protect it. Conversely, Pythia's goal is to enable system integrators verify that 3PIP vendors possess an IP without having access to it (i.e., protecting the privacy of the IP).

Different approaches against IP theft include obfuscation (by inserting additional gates into the design to hide the implementation) [2], watermarking (by embedding a signature in the design) [24], and fingerprinting (by embedding the buyer's signature to track the source of piracy) [25]. However, all of the aforementioned techniques tamper with the IP design and consider that verification takes place after the IP is outsourced to the integrator. Conversely, Pythia does not alter the circuit implementation and proves the functionality of the IP design while protecting its privacy.

VI. Concluding Remarks

In this paper, we present Pythia, a novel framework for privacy-preserving 3PIP verification. In our methodology, we transform 3PIP netlists into a zero knowledge-friendly format that can be used by Pythia's state machine to evaluate circuits. Pythia's back-end utilizes libSTARK to attest computational integrity of 3PIP circuit evaluation, proving knowledge of the IP to a system integrator without disclosing the netlist. To minimize the computational cost of zero-knowledge proofs, Pythia implements various optimizations to reduce the storage requirements for intermediate wires and exploit parallelism. In our experiments, we verify all ISCAS'85 benchmarks using at most 9 64-bit blocks for state storage, and efficiently utilize 96 cores to parallelize verification.

References

[1] 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.

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

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

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

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

[6] 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.

[7] 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.

[8] Y. Jin and Y. Makris, "Proof carrying-based information flow tracking for data secrecy protection and hardware trust," in *VLSI Test Symposium (VTS)*. IEEE, 2012, pp. 252–257.

[9] G. Moretti *et al.*, "Your Core – My Problem? Integration and Verification of IP," in *DAC*. ACM, 2001, pp. 170–171.

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

[11] E. Ben-Sasson *et al.*, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[12] 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.

[13] E. Ben-Sasson *et al.*, "SNARKs for C: Verifying program executions succinctly and in zero knowledge," in *CRYPTO*. Springer, 2013, pp. 90–108.

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

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

[16] R. Beaulieu *et al.*, "The SIMON and SPECK lightweight block ciphers," in *DAC*. IEEE, 2015, pp. 1–6.

[17] M. N. Wegman and J. L. Carter, "New hash functions and their use in authentication and set equality," *Journal of computer and system sciences*, vol. 22, no. 3, pp. 265–279, 1981.

[18] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," Cryptology ePrint Archive, 2004.

[19] T. Krovetz, "Message authentication on 64-bit architectures," in *Selected Areas in Cryptography (SAC)*. Springer, 2006, pp. 327–341.

[20] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 98–105.

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

[22] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE TIFS*, vol. 7, no. 1, pp. 25–40, 2011.

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

[24] A. B. Kahng *et al.*, "Watermarking techniques for intellectual property protection," in *DAC*. ACM, 1998, pp. 776–781.

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