Performance Analysis of Zero-Knowledge Proofs

Saichand Samudrala†, Jiawen Wu†, Chen Chen†, Haoxuan Shan*, Jonathan Ku*,
Yiran Chen*, Jeyavijayan Rajendran†
†*Texas A&M University*, **Duke University*
†{saichands, realjw, chenc, jv.rajendran}@tamu.edu,
*{haoxuan.shan, jonathan.ku, yiran.chen}@duke.edu

***Abstract*—Data privacy has become a severe concern due to the substantial growth in data gathering and processing, driven by the widespread adoption of cloud computing and cryptocurrency. Zero-Knowledge Proof (ZKP) has emerged as a promising cryptographic protocol for ensuring data privacy. However, ZKP suffers from high computational costs, making it excessively slow when implemented in software. To identify performance bottlenecks in the ZKP protocol, existing research has focused on CPU performance evaluation at the architecture level, considering factors such as execution time and memory consumption. Nevertheless, ZKP protocols have distinct memory and computing needs at each stage, these studies lack detailed CPU performance analyses necessary to improve ZKP performance and promote wider adoption.**

In this paper, we provide a comprehensive performance analysis of widely used ZKP libraries on CPUs. We perform four different analyses to characterize the CPU microarchitecture, memory, code, and scalability performance of the ZKP protocol on different CPUs. Our analysis reveals that the ZKP protocol performs differently on different CPUs, emphasizing the need for microarchitecture-specific analysis and optimizations. We evaluate loads and stores, last level cache (LLC) misses per kilo instructions (MPKI), and maximum memory bandwidth utilization of ZKP. We observe the *proving* stage consumes the highest memory bandwidth (25*Gbps*). We identify the functions that consume the most CPU time in each stage of the ZKP protocol and categorize each stage as compute, control-flow, or data-flow intensive. Also, we identify the *proving* stage as the most scalable with the potential to be sped up by parallel computation. These findings provide valuable insights for designing future ZKP accelerators, leading to more efficient and scalable ZKP implementations.

I. Introduction

Data privacy has become a significant concern due to the exponential surge in data collection and processing. The public administration industry experienced 495 reported data breach incidents with confirmed data loss between 2021 and 2022 [52]. According to IBM's report [37], the global average cost of a single data breach in 2023 was USD 4.45 million, reflecting a 15% over three years. To address this concern, privacy-preserving computation technologies [26] [25] have emerged in recent years, which enable data analysis and processing while ensuring that sensitive information remains confidential and private [11].

Zero-Knowledge Proof (ZKP) is one of the promising privacy-preserving computation technologies, gaining traction across industries due to its effectiveness and efficiency. Companies like IBM and Microsoft have integrated ZKP into numerous applications [36], [48]. IBM has focused on ZKP protocols for shorter, simpler, and more general applications [46]. Microsoft develops a new ZKP protocol that enhances user privacy and security for digital credential systems [48]. Scroll [4], a ZKP-based scaling solution, improves the transaction privacy of Ethereum, the world's second-largest cryptocurrency [15].

ZKP allows one party to prove to another party that they possess certain information without revealing the information [27]. However, ZKP entails large computation overheads. ZKP uses a custom representation of high-level programs, called constraint systems, to generate proofs [10]. The size of these constraint systems is several million times bigger than the original inputs. For example, representing a linear programming input with three equations and three variables requires over a million constraints due to the current ZKP protocol [10]. Further, generating proofs from these constraint systems requires modular operations with large bit-width ($\geq$ 256 bits) [59]. Due to these computation overheads, the proof generation of ZKP is time-consuming, taking up to an hour to generate a proof [12], [22], [59]. This translates to millions of hours for a server processing millions of transactions, highlighting the need to improve the computation efficiency of ZKP.

Several works accelerate a part of ZKP protocol on hardware platforms, such as CPU [55], ASIC [45], [50], [59], GPU [41], [47], and FPGA [6], [7], [53], [56], to improve its performance. However, these works are limited to improving one or two time-consuming modules of the ZKP protocol. Consequently, the overall performance improvement of the targeted ZKP protocol is low compared to the performance gains of the accelerated modules. For example, *PipeZK* [59] improves the performance of two modules, *Multi-Scalar Multiplication and Polynomial Multiplication*, achieving around 200$\times$ speedup for them. However, *PipeZK* achieves only a 5$\times$ overall speedup for the targeted ZKP protocol. The absence of a comprehensive analysis of the entire ZKP protocol impedes the development of hardware accelerators that enhance multiple modules to improve overall performance. Additionally, given the flexibility and accessibility of CPUs, there is a pressing need for a thorough performance analysis of the ZKP protocol on CPUs.

Recent works evaluate the performance of ZKP protocols at the architecture-level by profiling the protocols implemented in different libraries and reporting results like execution time and maximum memory consumption [19], [44]. However, the

works lack the microarchitecture-level analysis of ZKPs on CPUs. Mitigating bottlenecks at the microarchitecture-level greatly improves application performance. For example, based on the stalls in the CPU pipeline due to the unavailability of computation units and data cache misses, Shu-Ting Wang et al. [54] achieves around 10× speedup by implementing hardware accelerators for five deep learning applications. Ahmad Yasin et al. [58] characterize Big Data Analytics (BDA) workload on a state-of-the-art cloud server using microarchitecture-level analysis, identifying inefficiencies such as cache misses and branch mispredictions that degrade performance on CPU. From this understanding, Ahmad Yasin et al. [58] achieves a 1.14× speedup by performing code optimizations, such as loop re-ordering and reducing class dependency, saving millions of dollars on the data center scale. Thus, detailed analyses are needed to gain deeper insights into the performance bottlenecks of the ZKP protocol.

In short, the existing analyses [19], [44] have the following limitations: (i) lack of analysis of the percentage of CPU cycles wasted across the CPU microarchitecture. (ii) limited analysis of memory performance. (iii) limited observations of the ZKP program composition, and (iv) lack of the scalability characterization of the ZKP protocol. To address these limitations: For each execution stage of the ZKP protocol, we (i) perform a top-down analysis [57] to identify the percentage of pipeline stalls when executing the ZKP protocol on CPUs, (ii) conduct an in-depth memory analysis, including memory access, cache misses, and maximum memory bandwidth, to characterize the memory performance of the ZKP protocol, (iii) perform code analysis at the function-level and the instruction-level to understand the code composition of the ZKP protocol, and (iv) conduct scalability analysis to identify the parallelism within the computations of the ZKP protocol.

In summary, our main contributions are:

- We provide the first microarchitecture analysis of the ZKP protocol on CPUs. We observe that the ZKP protocol performs variably across different CPUs. For example, the *proving* stage is front-end bounded on Intel i7 and back-end bounded on Intel i9 CPUs. Consequently, evaluating its execution time on CPUs is inadequate, necessitating specific analysis and optimizations tailored to each CPU microarchitecture stage.
- We conduct memory analysis to illustrate loads and stores, the highest MPKIs of 1.03 and 0.48 of *witness* and *proving* stages, respectively, and the maximum memory bandwidth (25*GBps*) of the *proving* stage.
- We perform code analysis to identify the most time-consuming functions and to categorize each stage as compute, control-flow, and data-flow intensive.
- We conduct a scalability analysis to quantify the parallelism of each stage of the ZKP protocol. We identify the *proving* stage has high parallelism (71.67%), indicating it can be significantly sped up through parallel processing, thereby enhancing the ZKP protocol's overall performance.

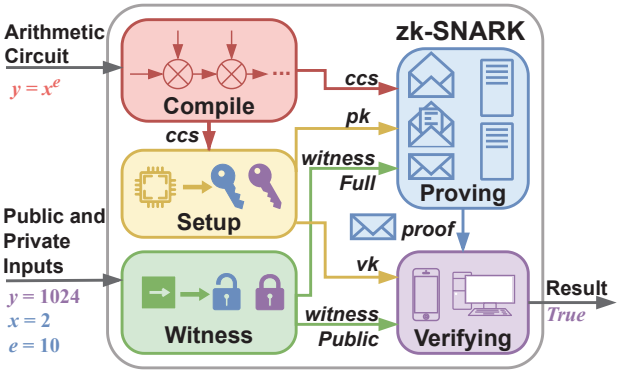

Fig. 1: The workflow of zk-SNARK. There are five stages involved in the zk-SNARK: *compile*, *setup*, *witness*, *proving*, and *verifying*.

II. Background

A. Zero-Knowledge Proofs

ZKPs [28] typically involves two parties: a prover who asserts a claim and a verifier who checks the validity of the claim. Prover generates a proof using both private and public inputs. Verifier validates the proof without knowing the private input and accepts the proof if it is correct. In other words, the prover aims to keep its private information confidential while proving the validity of a statement without revealing any additional information.

ZKP has three properties: (i) *Completeness:* If the statement is true and both parties follow the protocol properly, the verifier will accept the proof from the prover. (ii) *Soundness:* Prover cannot generate incorrect proof that the verifier accepts if the statement is false. (iii) *Zero-knowledge:* The protocol gives the verifier only the information that the statement is valid and does not leak any private data of the prover. We assume a malicious prover may attempt to cheat by providing false information or trying to convince the receiver of a false statement. The verifier can confirm the statement's correctness with a negligible probability that the prover is lying due to the *soundness* property. We also assume a malicious verifier may attempt to extract secret information from the statement provided by the prover. The verifier cannot infer any knowledge from the statement the prover sends due to the *zero-knowledge* property.

ZKPs can be categorized into two types based on their protocols: interactive and non-interactive. Interactive ZKP [21] requires a series of interactions between the prover and the verifier to correctly perform the proof of knowledge. The main advantage of interactive ZKP is their flexibility in handling various types of information. However, interactive ZKP can be time-consuming and require significant computational resources, limiting their applicability. Non-interactive ZKP, on the other hand, eliminates the need for interaction between the prover and the verifier. One of the most popular non-interactive ZKP is zk-SNARK (Zero-Knowledge Succinct Non-Interactive Argument of Knowledge) [13]. zk-SNARK is highly

efficient, generating small proof (in the order of hundreds of bytes) that can be verified in a few milliseconds. This efficiency makes zk-SNARKs a promising variant for both real-world applications and academic research. An example of their practical application is the Zcash cryptocurrency [33]. Thus, we will focus on zk-SNARK for the remainder of this paper.

B. zk-SNARK Workflow

Figure 1 illustrates the workflow of zk-SNARK, which takes inputs and arithmetic *circuits* to generate a proof and outputs *true* if the proof is correct. An arithmetic *circuit* [24] is a secure mathematical representation used in the zk-SNARK protocol (see Section II-C). zk-SNARK contains five stages: *compile*, *setup*, *witness*, *proving*, and *verifying*. The first three stages generate ZKP parameters essential for the effectiveness of the zk-SNARK protocol.

Compile takes the *circuit* and compiles it into a *compiled constraint system* (*ccs*), an intermediate representation of the *circuit*. In general, the stage converts a given computational statement into a mathematical form. In real-world applications, this stage is usually precomputed by a third party and provided to the users.

Setup generates ZKP parameters, including cryptographic keys, such as *proving key* (*pk*) and *verification key* (*vk*). *pk* is used by the *proving* stage to create succinct proofs for inputs the provers want to prove. *vk* is used by the *verifying* stage to verify the validity of proofs generated by the *proving* stage.

Witness receives public and private inputs from the prover, structuring them into a *witnessFull* without revealing sensitive data. *witnessFull* is used to generate proof for a statement containing all the necessary information to prove its validity. *witnessPublic* is a subset of the *witnessFull* and is for the verifier to verify the proof. It can be made public without compromising the privacy of the inputs.

Proving takes *ccs*, *pk*, and *witnessFull* from previous stages to generate a proof using cryptographic algorithms based on the zk-SNARK protocol. The proof is designed to be both succinct and non-interactive, allowing for efficient transmission and verification without further interaction.

Verifying uses parameters *vk* and *witnessPublic* to verify the correctness of the proof. If the proof is correct, *verifying* accepts the statement as *true* without learning any information beyond the proof's validity.

Elliptic curves (ECs) are the fundamental mathematical framework for ZKP. These curves, defined over finite fields, are the basis of the computations for all the stages of the zk-SNARK protocol. Elliptic curve operations facilitate efficient and secure computations. Similar to RSA, Elliptic curve cryptography (ECC) provides a similar level of security with much smaller key sizes. For instance, a 256-bit key in ECC provides comparable security to a 3072-bit key in RSA [8]. This smaller key size results in faster computations and reduced storage requirements [8]. However, there is still a trade-off: Using elliptic curves with larger bit widths can increase security but also introduce greater computational overhead. Popular elliptic curves used in the zk-SNARK protocol include the BN128 and the BLS12-381. For example, Zcash has transitioned from using the BN128 to the BLS12-381 curve to enhance its security [33].

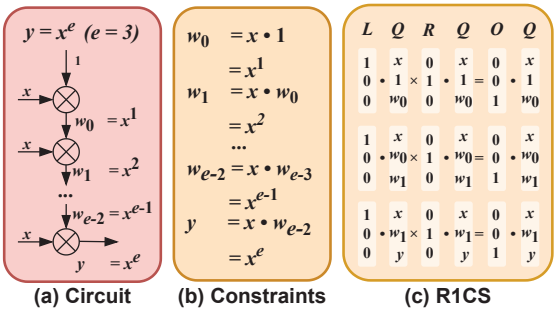

Fig. 2: This example illustrates how the computational circuit ($y = x^e$, where $e = 3$) is compiled into the R1CS representation. (a) Shows the *circuit*, and irepresents the *circuit* into a set of constraints (b), and then compiles the *circuit* into R1CS representation (c).

C. Compiling Arithmetic Circuits

In the zk-SNARK protocol, compiling an arithmetic *circuit* to constraints is a crucial step that forms the foundation for the entire workflow. The *compile* stage generates a set of constraints based on the *circuit* and converts these constraints into a form called Rank-1 Constraint System (R1CS) for efficient computation [34].

An arithmetic *circuit* is a mathematical construct in zk-SNARK for generating and verifying proofs. Unlike traditional Boolean circuits, which deal with binary values, arithmetic *circuits* process numerical values. An arithmetic *circuit* converts a polynomial function into an acyclic graph of *gates* for addition and multiplication. Each *gate* contains two inputs and one output. Figure 2 (a) shows the *circuit* example for the polynomial function $y = x^3$. Since the function only includes multiplication operations, the *circuit* contains three multiplication *gates* with three x inputs, one y output, and two intermediate values w_0 and w_1.

For each *gate* of the *circuit*, the stage will generate an associated constraint as shown in Figure 2(b). These constraints specify the relationships among the variables, ensuring the validity of the statement. For the example *circuit*, the associated constraints are $\{w_0 = x \cdot 1,\ w_1 = x \cdot w_0,\ y = x \cdot w_1\}$.

The *compile* stage converts the constraints into the R1CS format [34], which involves: (i) variables: representing inputs, outputs, and intermediary values of the computational problem. (ii) R1CS constraints: equations expressing relationships among the variables, enabling efficient computation due to their linear nature. (iii) linear combinations: The constraints involve linear combinations of variables represented as a vector dot product, facilitating computational efficiency. The R1CS format, depicted in Figure 2(c), represents variables and their linear combinations as vectors. The vectors L, R, and O represent the coefficients in the left input, right input,

and output of a *gate*, respectively, while Q represents the coefficients of variables in the quadratic term [34]. Given the associated constraints of the arithmetic *circuit* $y = x^3$, the third constraint $y = x \cdot w_1$ in the R1CS format is: $L = [1, 0, 0]$, $R = [0, 1, 0]$, $O = [0, 0, 1]$, $Q = [x, w_1, y]$. The R1CS constraint $L \cdot Q \times R \cdot Q = O \cdot Q$ can be interpreted as $x \cdot w_1 = y$.

III. Methodology

ZKP performance is critical for its use in privacy-preserving applications. However, the lack of a thorough performance analysis limits the development of efficient ZKP applications. Existing works on analysis provide details like execution time, maximum memory consumption, and proof size of zk-SNARKs when running on CPUs [19], [44]. But the previous works have these following limitations.

L1. Lack of analysis of the percentage of CPU cycles wasted across the CPU microarchitecture. From the current analysis, although we know the overall execution time performance of the zk-SNARK protocol, the performance at each stage of the CPU is unclear, and its understanding is crucial to mitigate bottlenecks at a microarchitecture-level. Moreover, since the performance of the stages is correlated, it is important to understand the CPU's performance hierarchically. For example, if the fetch and decode stage is undersupplying instructions, the execution units stall. In this scenario, the fetch and decode stage needs optimization to improve the execution performance. Naively adding more execution units will not improve the performance. *Thus, a finer microarchitecture analysis is needed for ZKP performance.*

L2. Limited analysis of memory performance. Prior work presents the maximum main memory occupied by zk-SNARK protocol when running on CPUs [19], [44]. However, memory accesses (i.e., load and store), last level cache (LLC) misses per kilo instructions (MPKI), and maximum bandwidth utilization are unclear. The time taken for data movement between the CPU and main memory introduces latency. Accessing data from main memory is slower than accessing data from CPU caches. This latency can cause delays in application execution, especially if the application frequently needs to access data from memory. Moreover, the memory bus and subsystem's bandwidth determines how much data can be transferred between the CPU and main memory within a given time frame. Insufficient bandwidth can lead to bottlenecks, where the CPU has to wait for data to be transferred, reducing overall performance. *Therefore, it is necessary to conduct a comprehensive memory analysis.*

L3. Limited observations of the ZKP program composition. Previous works [19], [44] do not provide analysis of ZKP code. A program comprises multiple functions, each serving a specific purpose. For instance, the malloc function assigns memory to the application as it runs. These functions utilize distinct CPU memory and time depending on their tasks. As per the code's structure, certain functions are executed more frequently than others. Recognizing and enhancing the latency performance of these functions can substantially improve the overall application performance. A software code comprises compute, control-flow, and data-flow instructions. Control-flow and data movement significantly impact the performance of applications in various ways [20]. For example, excessive branching of complex conditional logic hinders performance due to branch mispredictions. Branch mispredictions can lead to pipeline stalls and reduced CPU throughput [51]. Further, data movement among registers, caches, and main memory of CPUs causes latency overhead. Excessive use of data movement operations degrades ZKP performance. *Hence, it is essential to perform a code analysis of the ZKP library to identify time-consuming functions and also for its control-flow and data-flow instructions.*

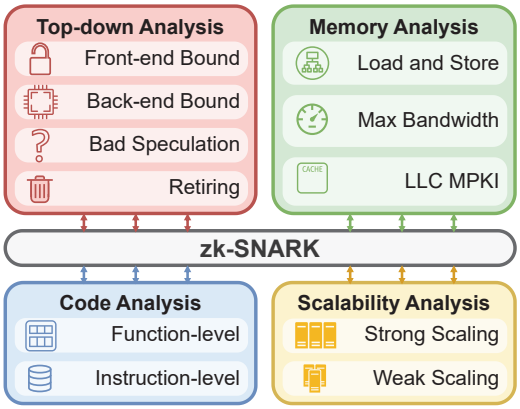

Fig. 3: A comprehensive performance analysis framework for zk-SNARKs, incorporating top-down analysis, memory analysis, code analysis, and scalability analysis.

L4. Lack of scalability characterization. Scalability is another problem for the widespread adoption of ZKP protocols in larger applications. Scalable algorithms and code can process increased workloads and larger datasets without significantly degrading performance [9], [31], [35]. This is important for applications deployed in environments with fluctuating demand or rapidly growing user bases [32]. Scalable software efficiently utilizes resources such as CPUs, allowing the application to make the most of the underlying hardware infrastructure. This can lead to cost savings and improved efficiency, especially in cloud computing environments where resource usage directly impacts operational expenses [32]. Previous works lack a scalability analysis of zk-SNARK libraries. *Therefore, an analysis is necessary to understand the scalability of the zk-SNARK protocol.*

In the following subsections, we present our approach to analyzing the performance of ZKP libraries, addressing the limitations (**L1-L4**) of the previous works [19], [44]. Figure 3 presents the analyses we performed on the ZKP protocol.

A. Top-down Microarchitecture Analysis

To address **L1**, we analyze the performance of zk-SNARK applications running on CPU to understand the bottlenecks at the microarchitecture-level [57]. Modern CPUs employ pipelining and techniques like multithreading, out-of-order

execution, and instruction-level parallelism to improve the utilization of computation units. Intel introduced the top-down microarchitecture analysis method, top-down analysis for short. It is a hierarchical organization of event-based metrics that identifies the performance bottlenecks in an application [38]. Top-down analysis splits pipeline slots of a CPU from top-level microarchitecture stages to lower-level compute units. It aims to show how well the CPU utilized the pipelines on average while running an application. The stages of microarchitecture-level can be broadly classified into four: instruction fetch and decode, speculative branch prediction, micro-operations issue and execution, and micro-operations retiring [38], [57].

Our evaluation includes the top-level metrics in the hierarchy of top-down microarchitecture analysis. Intel classified pipeline slots into these four categories: front-end bound, bad speculation, back-end bound, and retiring [38], which offers a comprehensive assessment of zk-SNARK protocol performance at the microarchitecture-level. **Front-end bound** refers to the fraction of CPU pipeline slots idle compared to the overall slots allotted for instruction fetch and decode. The front-end is the first CPU core unit responsible for fetching and decoding instructions. **Back-end bound** is the fraction of CPU pipeline slots during which instructions are not issued to execution units due to the lack of resources. The back-end of the CPU is the portion that delivers the decoded instructions to their respective execution units. **Bad speculation** refers to the fraction of pipeline slots wasted caused by branch mispredictions, and the CPU is busy fetching and executing non-useful operations. **Retiring** denotes the fraction of CPU pipeline slots used to retire instructions successfully over the allocated slots. It is ideal for a program to have a high percentage of pipeline slots classified in this category.

Top-down analysis has been widely deployed to identify the performance bottleneck of industrial applications like cloud computing and deep learning [54], [58]. We employ these widely used metrics to determine the inefficiencies in executing the instructions about the zk-SNARK protocol on the CPU at the microarchitecture-level. We run each stage of the zk-SNARK protocol separately and perform the analysis for various sizes of *circuits*.

B. Memory Analysis

To address **L2**, we evaluate the memory performance, including loads and stores, LLC load MPKI, and maximum bandwidth utilization. We measure the LLC load MPKI, which is a standard method used in estimating the cache performance of an application since LLC misses have a larger latency compared to L1 and L2 caches [16], [18], [39], [40]. This analysis is critical in determining the overall latency of ZKP applications running on a CPU. Load and store operations moving data between CPU and DRAM memory take hundreds of CPU clock cycles and can stall CPU executing instructions. CPUs deploy small, fast caches to store frequently used data to avoid these time-consuming data movements. However, depending on the spatial and temporal locality of the application's data, cache misses occur, leading to access to DRAM for the data. Further, the maximum rate at which the zk-SNARK protocol accesses the data from DRAM is measured by the maximum bandwidth utilization. Inadequate memory bandwidth can cause bottlenecks, forcing the CPU to wait for data transfer and lowering overall performance. To determine the number of loads, stores, and maximum bandwidth utilization of each stage of the ZKP protocol, we run a memory access analysis of the ZKP protocol using the Intel *VTune Profiler* [38]. We use the number of instructions and cache misses provided by the *perf* tool [1] to determine MPKI. We report the highest MPKI values across the different constraint sizes to consider the worst-case scenario.

C. Code Analysis

To address **L3**, we categorize code analysis into function-level and instruction-level analysis. In functional-level analysis, we determine the functions in the ZKP library code that significantly consume the CPU time using the Intel *VTune Profiler* [38]. In instruction-level analysis, we classify instruction streams based on operations. Compute operations involve arithmetic opcodes like *add*, *and*, etc. Control-Flow instructions alter program execution flow with operations like *jz*, *jnb*, etc. Data-Flow instructions manage data movement between registers and memory, including operations like *mov*, *push*, etc. We determine the distribution of compute, control, and data flow instructions in each stage of the ZKP protocol, profiling the instruction opcode mix in the selected ZKP libraries using the *DynamoRIO* tool [2]. With the opcode mix data, we can find the dominating type of instructions to categorize each stage of the ZKP protocol into compute, control-flow, and data-flow intensive. This analysis aids in comprehending the underlying causes of the potential bottlenecks and devising strategies for performance optimization.

D. Scalability Analysis

Scalability analysis encompasses strong scaling (SS) and weak scaling (WS) methodologies, each serving distinct purposes in evaluating system performance. fdBoth SS and WS analyses are critical for understanding the limitations of a system as it scales, aiding in the design and optimization of distributed computing architectures for ZKP [9], [31].

Based on the Amdahl's law, SS evaluates how efficiently a system's performance as the number of threads increases while keeping the problem size fixed. It provides insights into the parallel efficiency of a system by measuring the speedup using Equation (1) achieved as threads are added [9].

$$Speedup_{SS} = t_1/t_n, \tag{1}$$

where $Speedup_{SS}$ is calculated using the baseline execution time t_1 for one thread and the execution time for n threads t_n on a CPU for a zk-SNARK protocol's stage for a *circuit*. In this work, we present the speedup of five stages of zk-SNARK when increasing the number of threads in the CPU and observing the speedup for different numbers of constraints.

Conversely, based on the Gustafson's law, WS assesses the system's ability to maintain a constant workload per thread as the problem size scales proportionally with the number of units [31]. The speedup of a system can be measured using the Equation (2):

$$Speedup_{WS} = t_1 \times sf/t_n, \quad (2)$$

where $Speedup_{WS}$ is calculated using the baseline execution time t_1 when running one thread and the execution time running n threads t_n on a CPU for a zk-SNARK protocol's stage for a *circuit*. The scaling factor *sf* is the increasing factor of constraint size compared to the size when running on one thread. We analyze the speedup as the number of threads and constraints double.

IV. Evaluation

In this section, we present the experimental results obtained from the performance characterization of the ZKP libraries for testbench arithmetic *circuits*. We employ the widely used, industry-standard Intel *VTune Profiler* [38], *perf* [1], and *DynamoRIO* [2] tools for the performance analysis of the zk-SNARK protocol[1]. Investigating CPU performance is crucial because CPUs are the most popular computing platform [42] for zk-SNARK. We perform a fine-grained analysis of the zk-SNARK protocol on different CPUs across different elliptic curves. We also characterized the zk-SNARK protocol into 5 stages, as mentioned in Figure 1. The CPUs and related specifications are shown in Table I. We configure hyperthreading for scalability analysis. For other experiments, we follow a one-thread-per-core binding model with `#threads==#cores`. We use Linux commands to manually adjust CPU cores and hyperthreading statuses for the required number of threads. We then ran experiments three times and considered the average values to mitigate the measurement noise.

A. Experimental Setup

Selection of zk-SNARK libraries. zk-SNARK protocols are implemented by several libraries [5], [14]. For analysis, we selected *circom* [3] and *snarkjs* [5] zk-SNARK libraries for the performance characterization of the zk-SNARK protocol. *circom* is for the *compile* stage, and *snarkjs* is for the rest of the four stages, including *setup*, *witness*, *proving*, and *verifying*. The combination of *circom* and *snarkjs* libraries is one of the most popular frameworks for the zk-SNARK protocol. *circom* is used for compiling arithmetic *circuits* to *compiled constraint systems* (*ccs*), as mentioned in Section II-C. *Circom* takes the arithmetic *circuits* and compiles them into the R1CS format. *Snarkjs* implements the rest of the zk-SNARK in JavaScript and pure WebAssembly [5]. It supports two proving schemes, the *Groth16* [30] and *PlonK* [23]. The *proving* time of *PlonK* is twice as slow compared to *Groth16*. *Groth16* has become a de-facto standard used in several blockchain projects due to its constant proof size and efficiency [14]. Therefore, we analyze the performance of the most popular proving scheme, *Groth16* [30]. By analyzing *Groth16*, we show our strategies offer insights to guide future designs. Designers can replicate our strategies for different ZKP programs. We used the latest versions of *circom* [3] and *snarkjs* [5] for the evaluation.

Selection of benchmarking circuits. We evaluate the performance of zk-SNARK with the popular proving scheme, *Groth16* [30], using arithmetic *circuit* that performs exponential calculations. The exponentiate *circuit* represents the arithmetic *circuit* for the exponentiation function that computes y such that $y = x^e$ for the given inputs x, e, as mentioned in Section II-C. In this case, the value of e equals the number of *constraints*, and three inputs, y, x, e, are used for *witness generation*. We considered the same *circuit* for commonly used exponentiate functions in ZKPs from previous works [19], [60]. We choose the number of *constraints* as one of the variables for our experiment to be consistent with previous work [19]. The number of *constraints* in zk-SNARK libraries is limited. For example, *Groth16* on BN128 cannot exceed 268 million *constraints* [5]. We choose the range of constraints from 2^{10} to 2^{18}. This covers the majority of the applications in real-world scenarios [19].

Selection of elliptic curves. *Circom* and *snarkjs* can be instantiated with different elliptic curves, including BN128 and BLS12-381 [5], with a trade-off between computation efficiency and security levels. We considered both BN128 and BLS12-381 for their widespread adoption and efficiency [15].

B. Experimental Results

Execution time analysis. We collected the elapsed times running each stage separately on the CPUs. Our results show that the *setup* (76.1%) is the most time-consuming stage, followed by the *proving* (13.4%) stage across all constraint sizes of the exponentiation circuit. Our execution time results are consistent with those of *zk-Bench* [19] results.

Top-down microarchitecture analysis. We aim to classify where pipeline stalls occur for each stage of the zk-SNARK protocol at the microarchitecture-level of the CPU. To achieve this, we conduct a top-down microarchitecture analysis of the *circom* and *snarkjs* zk-SNARK libraries running on various CPUs, as discussed in Section III-A. Figure 4 presents the percentage of where the pipeline slots in each of the four categories for each stage of the ZKP protocol across different constraint sizes of the exponentiation *circuits*. Figure 4 shows that BN128 and BLS12-381 elliptic curves have similar results across different stages of the zk-SNARK protocol and CPUs.

We observe distinct behaviors of where the pipeline slots are classified for *compile*, *setup*, and *proving* stages across different CPUs. In the *compile* stage, pipeline slots are in the back-end bound category for more than 90% of the constraint sizes on the i5 and the i9 CPUs and are in the front-end bound category on the i7 CPU. The *setup* stage is mostly classified in the front-end bound category on the i7 CPU, in the bad speculation category on the i5 CPU, and in the back-end bound category on the i9 CPUs. The *proving* stage is in the front-end bound category for 60% of the constraint sizes on the i7 CPU, both in the back-end bound and bad speculation

[1]https://github.com/Saichand-Samudrala/Performance-Analysis-of-Zero-Knowledge-Proofs

TABLE I: Hardware configuration of the experimental setup. (Number of performance cores (#Cores (Perf)), number of efficient cores (#Cores (Eff)), number of SMT threads (#SMT), DRAM type (Type), number of DRAM channels (#DRAM Ch), DRAM maximum memory bandwidth (Mem BW), last level cache (LLC), version of nodejs (nodejs).)

CPU	OS	#Cores (Perf)	#Cores (Eff)	#SMT	DRAM	Type	#DRAM Ch	Mem BW	LLC	nodejs
Intel i7-8650U	Ubuntu 22.04 LTS	4	0	8	16 GB	LPDDR3	2	34.1 GB/s	8 MiB	v12.22.9
Intel i5-11400	Ubuntu 24.04 LTS	6	0	12	8 GB	DDR4	1	17.0 GB/s	12 MiB	v18.19.1
Intel i9-13900K	Ubuntu 22.04 LTS	8	16	32	32 GB	DDR5	4	89.6 GB/s	36 MiB	v22.2.0

category for 60% of the constraint sizes on the i5 CPU, and in the back-end bound category on the i9 CPU. The results also show that most pipeline slots in *witness* and *verifying* stages are in the front-end bound category across the CPUs. This indicates the implementation is leading to fetch and decode, branch speculations, and parallel processing inefficiencies [57], resulting in pipeline stalls in the CPUs.

Key Takeaway 1. *Top-down analysis reveals different ZKP performances on different CPUs. Thus, only analyzing the execution time performance of the ZKP protocol on CPUs is insufficient and emphasizes the need for finer microarchitecture-specific analysis and optimizations.*

The bottleneck related to fetch and decode makes it challenging to supply the required number of instructions to the back-end execution units in the CPUs. Branch mispredictions lead to flushing the pipelines. Limited parallelism in the ZKP protocol implementation leads to suboptimal utilization of multi-core CPUs. Hence, naively increasing the number of computation units will not improve the latency performance. Improving prefetching strategies and optimizing branch prediction algorithms for the ZKP protocol can address front-end bound and bad speculation problems. Further, enhancing the ZKP protocol to maximize parallelism improves the performance on multi-core CPUs.

Memory analysis. To evaluate the memory performance of each stage of the zk-SNARK protocol, we obtain loads and stores, LLC MPKI, and maximum memory bandwidth utilization on three different CPUs. Figure 5 presents the loads and stores of each stage of the zk-SNARK protocol for nine different constraint sizes of the exponentiation *circuits*.

We observe from the results that the *setup* stage and *proving* stage respectively require around 1000× and 100× the number of loads than the *witness* and the *verifying* stage. Loads and stores for the *verifying* stage and the *witness* stage remained constant and only slightly varied across different numbers of constraints. The number of loads and stores have similar trends in most stages, with one major difference being that the *setup* stage has around 10× more loads than stores. Table II shows the maximum LLC MPKI for each stage of the zk-SNARK protocol across the different constraint sizes. From Table II we observe the *witness* and *proving* stages have the highest MPKIs of 1.03 and 0.48 respectively, and the *setup* stage has the lowest MPKI of 0.06 among all the stages.

Table III shows the average maximum memory bandwidth requirements of each stage of the zk-SNARK protocol averaged over the nine different constraint sizes of the exponentiation *circuits*. The *proving* and *setup* stages require the highest maximum memory bandwidth of 25*GBps* and 23.4*GBps* respectively, around 2× compared to the *compile* stage.

TABLE II: Memory analysis: LLC MPKI for the five stages of zk-SNARK for the exponentiation *circuits*.

Stages	i7-BN	i7-BLS	i5-BN	i5-BLS	i9-BN	i9-BLS
compile	0.32	0.34	0.32	0.22	0.18	0.22
setup	0.04	0.03	0.08	0.06	0.05	0.03
witness	0.62	0.47	0.28	0.40	0.29	1.03
proving	0.17	0.14	0.48	0.34	0.45	0.28
verifying	0.15	0.10	0.20	0.16	0.15	0.15

TABLE III: Memory analysis: Maximum memory bandwidth for the five stages of zk-SNARK for the exponentiation *circuits*. The average values for three different CPUs.

EC	*compile*	*setup*	*witness*	*proving*	*verifying*
BN (GBps)	10.30	23.40	2.70	25.00	5.20
BLS (GBps)	11.50	20.20	2.80	22.90	4.40

Key Takeaway 2. *Memory analysis shows that the proving stage consumes the highest memory bandwidth (25 GBps).* Analyzing and optimizing the memory access patterns, leveraging the input sparsity, adopting data compression techniques [29], and using memory-efficient designs such as HAAC [49], a cryptographic protocol accelerator, can enhance memory performance.

Code analysis. We perform analysis at the function-level and instruction-level for each stage of the ZKP protocol. In function-level analysis, we identify the functions within the ZKP library code that heavily utilize CPU time using the Intel *VTune Profiler* [38]. In instruction-level analysis, we examine the distribution of compute, control-flow, and data-flow instructions using the *DynamoRIO* tool [2].

TABLE IV: Code analysis: Identify time-consuming functions.

Function	Description
memcpy	Copies a block of data to another address.
bigint	Performs calculations on large integers.
heap allocation	Manages the allocation of dynamic memory.
malloc	Manages the allocation of dynamic memory.
page fault exception handler	Handles page faults and retrieves the data.

For function-level analysis, Table IV shows that big integer (>256-bit) computation (*bigint*), dynamic memory allocation (*malloc*), and data flow (*memcpy*) are significant functions

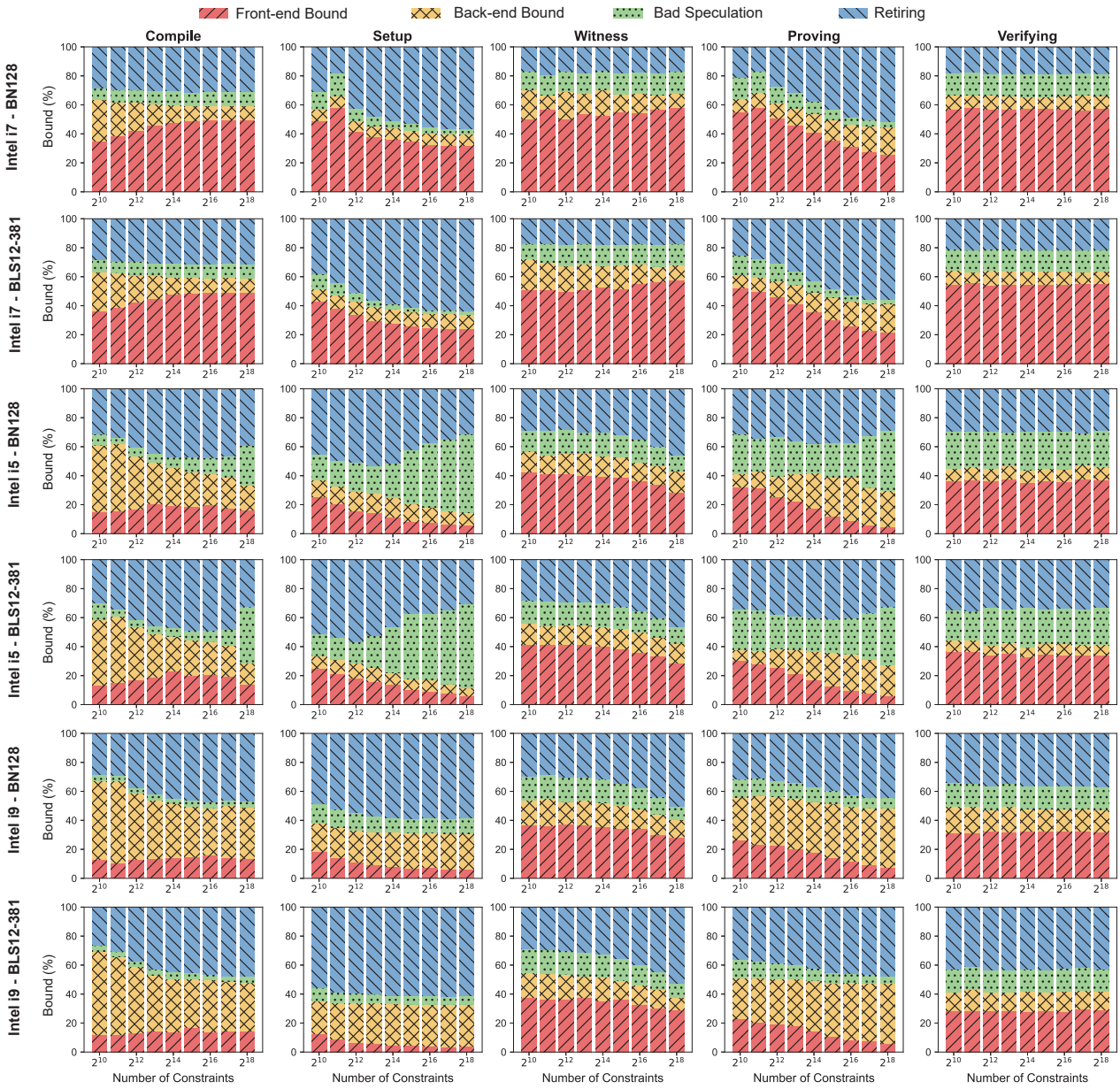

Fig. 4: Top-down microarchitecture analysis for the five stages of zk-SNARK for the exponentiation *circuits*.

occupying the CPU time. We observe from the results that in the *compile* stage, *malloc* occupies around 12%, *memcpy* occupies around 8%, and *bigint* occupies 5% of the CPU time, respectively. In the *proving* stage, *memcpy* and in the *verifying* stage, *bigint* each occupy 10% of the CPU time.

Key Takeaway 3. *Code analysis identifies functions related to both generic functions like Malloc and application-specific functions like bigint that dominate the CPU execution time. bigint* can be optimized in CPUs by changing representations to such as the Chinese Remainder Theorem (CRT), which converts *bigint* numbers to a set of int numbers, increasing parallel computation. Moreover, designers can implement a hardware accelerator for CRT [42], [43] and hardware/software co-design [17] to efficiently process the large data of *bigint*.

Further, we analyze each stage of the zk-SNARK protocol at the instruction-level using the *DynamoRIO* tool [2] to classify each stage into compute, control-flow, and data-flow intensive stages. For instruction-level analysis, Table V shows that *setup*, *proving*, *verifying* stages are compute-intensive, occupying 42.56%, 47.31%, and 48.20%, on average respectively. The *compile* stage is data-flow intensive with 39.61% of occupancy. The *witness* stage is control-flow intensive compared to the rest of the stages. The results indicate the need for the development of libraries that optimize the identified

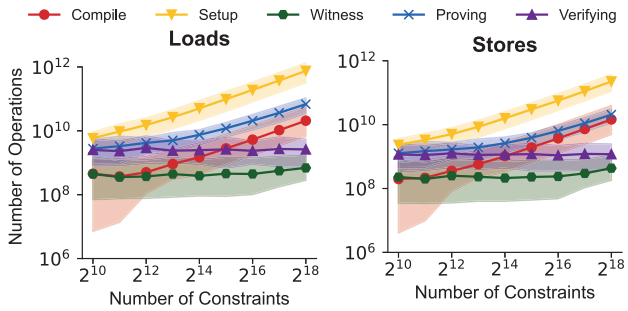

Fig. 5: Memory analysis: A breakdown of loads and stores for the five stages of zk-SNARK for the exponentiation *circuits*. The middle line shows the average values, while the maximum and minimum values are filled in based on data from three different CPUs and two elliptic curves.

TABLE V: Code analysis: The average percentages of different opcode types on BN128 and BLS12-381 elliptic curves across five stages of zk-SNARK for the exponentiation *circuits*. Opcode types include compute (Comp), control-flow (Ctrl), and data-flow (data).

Stages	BN			BLS		
	Comp (%)	Ctrl (%)	Data (%)	Comp (%)	Ctrl (%)	Data (%)
compile	32.68	28.99	38.33	38.68	20.42	40.89
setup	42.60	20.16	37.24	42.53	20.36	37.10
witness	35.96	29.49	34.55	39.16	28.26	32.57
proving	40.96	22.69	36.35	53.66	16.27	30.07
verifying	46.66	24.81	28.53	49.75	23.04	27.21

time-consuming functions. Furthermore, accelerating each stage based on compute, control, and data flow categories can enhance the latency performance of the ZKP protocol.

Key Takeaway 4. *Instruction-level analysis shows that the proving stage has more than 30% data-movement operations.* Designers can use novel architecture such as process-in-memory (PIM) [45] to reduce the latency of data movement between memory components.

Scalability analysis. We evaluate strong scaling (SS) and weak scaling (WS) to determine the scalability of the zk-SNARK protocol. Figure 6 shows the values of $Speedup_{SS}$ as the number of CPU threads increases while the number of constraints remains fixed for nine different sizes of the constraints on the i9 CPU. The results show that for the largest constraint size 2^{18}, the *setup* and the *proving* stages have on average 5.26× and 3.51× speedup, respectively, compared to the rest of the stages. The speedup of the *compile* stage and the *witness* stage increases initially as the number of threads increases and then saturates. For example, from Figure 6, for the constraint 2^{11}, the speedup of the *compile* stage and the *witness* stage saturate at around two, showing the maximal speedup the two stages can gain from parallelization.

The speedup of the *verifying* stage remained constant independent of the constraint size. This result correlates to the constant execution time of the *verifying* stage independent of the constraint size. In some results, such as during the *compile* stage with a constraint size of 2^{10}, the speedup achieved with 24 threads is lower than that with 18 threads. This is due to the short execution time (< one second) of the task, where the overhead of managing additional threads and profiling measurement becomes comparable to the actual computation time, leading to diminished performance gains.

Figure 7 shows the $Speedup_{WS}$ for each stage of the ZKP protocol as both the number of threads and constraints double with threads running from 1 to 32 and the constraints running from 2^{13} to 2^{18}. Figure 7 shows that as the constraint size increases, *proving* stage is more scalable than the *setup* and *compile* stages. The *witness* and the *verifying* stages show an approximately linear increment of speedup since both the execution time when running on one thread, t_1, and on n threads, t_n, in $Speedup_{WS}$ are approximately constant. This is because the *witness* generation time and *verifying* time are independent of the constraint size, and the scaling factor increases at the same rate as the increment in the number of threads.

Further, to quantify the scalability, we determine the parallelism in the ZKP protocol. We obtain the serial and parallel parts in each stage of the zk-SNARK protocol by fitting the SS and WS results to Amdhal's [9] and Gustafson's [31] formulae, respectively. Table VI shows the average values of serial and parallel parts averaged for nine different constraint sizes for all the stages of the zk-SNARK protocol. The results show that the *proving* stage (71.67%) has a higher parallelism compared to the *compile* (34.44%) and the *setup* (31.40%) stages. WS results show that the *proving* has around 3× parallelism compared to *compile* and *setup*. The *witness* and the *verifying* stages show more than 90% parallelism for WS since both t_1 and t_n in $Speedup_{WS}$ are approximately constant because the witness generation and proof verification times are independent of the constraint size, and the scaling factor increases the same as the increment in the number of threads.

The results indicate that *proving* stage can be sped up by parallel processing, thereby potentially reducing the computational time of the ZKP protocol. WS's results demonstrate that over 90% parallelism for both the *witness* and the *verifying* stages compared to SS results due to their constant time execution independent of the size of the constraints. However, the speedup saturates for both the *witness* and the *verifying* stages. For example, with the largest constraint 2^{18}, speedup saturates with fewer than six threads, highlighting the dominance of the serial part. Unlike these stages, the speedup of the *proving* stage does not saturate after 24 threads.

Key Takeaway 5. *Scalability analysis shows that the proving stage has 3× parallelism compared to the rest.* Heterogenous hardware, providing high parallelism, such as GPU, can be used to improve the performance of the *proving* stage.

V. Related Work

Several works, including *PipeZK* [59] and *DistMSM* [41], accelerate the ZKP protocol through various hardware platforms. *PipeZK* [59], an ASIC-based hardware accelerator, proposes a

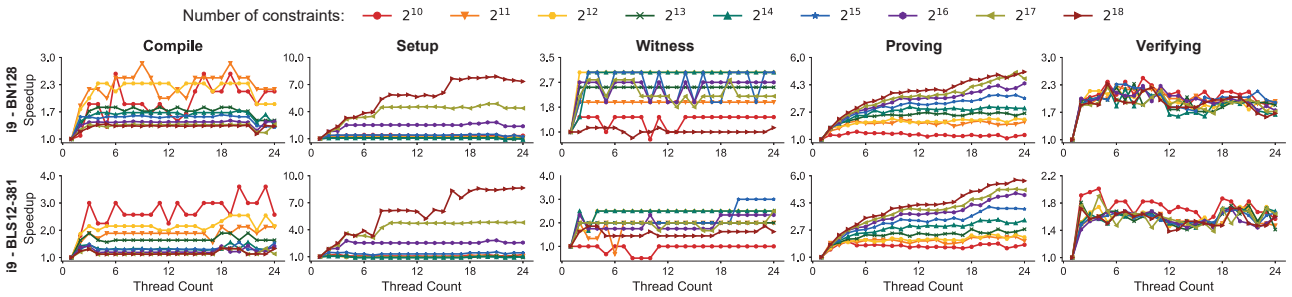

Fig. 6: Scalability analysis: Strong scaling analysis for the five stages of zk-SNARK for the exponentiation *circuits*.

TABLE VI: Scalability analysis: Serial and parallel percentage across five stages of zk-SNARK for the exponentiation *circuits*.

Stage	SS-i9-BN		SS-i9-BLS		WS-i9-BN		WS-i9-BLS	
	Serial (%)	Parallel (%)	Serial (%)	Parallel (%)	Serial (%)	Parallel (%)	Serial (%)	Parallel (%)
compile	58.09	41.90	62.50	37.49	69.65	30.35	71.98	28.02
setup	41.35	58.64	68.30	31.69	73.59	26.41	75.11	24.89
witness	31.73	68.26	50.17	49.82	3.59	96.41	7.75	92.25
proving	27.28	72.71	31.06	68.93	29.57	70.43	25.38	74.62
verifying	43.68	56.31	57.56	42.43	1.00	99.00	1.00	99.00

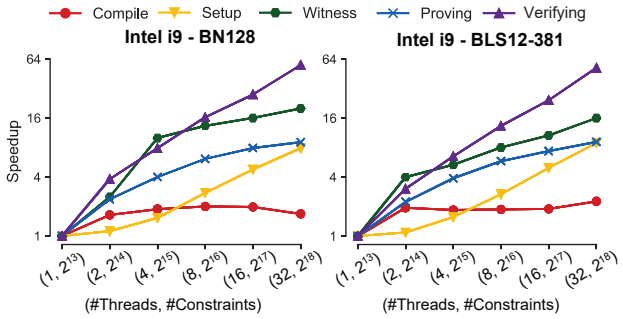

Fig. 7: Scalability analysis: Weak scaling analysis for the five stages of zk-SNARK for the exponentiation *circuits* for i9 CPU. The thread count doubles from 1 to 32, while the number of constraints doubles from 2^{13} to 2^{18} correspondingly.

hardware architecture for two major time-consuming modules in zk-SNARK protocol. *PipeZK* implements bandwidth-efficient hardware using the pipelining techniques. Further, the paper employs a dynamic work dispatch mechanism to share computation units to execute operations efficiently. This work achieves the highest speedup of around 200× for one of the optimized modules. However, only 5× speedup is observed for that targeted ZKP application. *DistMSM* [41], a multi-GPU hardware accelerator, accelerates a time-consuming module of the zk-SNARK protocol. This paper performs a workload analysis at the GPU thread level for the time-consuming zk-SNARK module. This work leverages the parallelism in the workload by distributing the workloads to multiple GPUs. Further, these works show that improving the performance of multiple components is important to enhancing the overall zk-SNARK protocol. However, the lack of analyses hinders the development of hardware accelerators to accelerate multiple components for higher overall performance.

zk-Bench [19] conducts performance analysis of zk-SNARK to assess the time taken for execution, memory consumption, and proof size of the protocols when on CPUs. Max Kobelt et al. [44] implements a configurable benchmarking framework to compare the performance of different zk-SNARK libraries. This work presents performance results such as execution time and memory consumption for the zk-SNARK protocol. In summary, the previous works provide analysis at the architecture level and have the following limitations: lack of CPU microarchitecture, memory performance, code, and scalability analyses, as discussed in Section III. zk-SNARK protocol consists of multiple stages, each with specific memory bandwidth and computing requirements. Existing works lack the analysis of these requirements in detail. However, our work conducts a detailed performance analysis of the zk-SNARK protocol at the micro-architecture level, supporting researchers in implementing efficient zk-SNARK applications.

VI. Conclusion

The lack of a comprehensive analysis of the entire Zero-Knowledge Proof (ZKP) protocol hinders the development of accelerators for the complete protocol. Our work conducts comprehensive performance analyses of the ZKP protocol at the microarchitecture-level. Our study reveals the varying performance of the ZKP protocol across different CPUs. Hence, evaluating its execution time on CPUs is insufficient and requires microarchitecture-specific analysis and optimizations. Additionally, we provide detailed analyses of memory performance, code composition, and scalability. The results provide comprehensive guidance for developers and researchers in selecting and optimizing ZKP solutions for diverse applications and computing environments.

References

[1] "Perf Tool," https://perf.wiki.kernel.org/, 2006, Last accessed on 06/03/2024.

[2] "DynamoRIO," https://dynamorio.org/, 2010, Last accessed on 06/03/2024.

[3] "circom," https://github.com/iden3/circom, 2016, Last accessed on 06/03/2024.

[4] "Scroll," https://scroll.io/, 2021, Last accessed on 06/03/2024.

[5] "snarkjs," https://github.com/iden3/snarkjs, 2021, Last accessed on 06/03/2024.

[6] K. Aasaraai, D. Beaver, E. Cesena, R. Maganti, N. Stalder, and J. Varela, "FPGA Acceleration of Multi-Scalar Multiplication: CycloneMSM," *Cryptology ePrint Archive*, 2022.

[7] K. Aasaraai, E. Cesena, R. Maganti, N. Stalder, J. Varela, and K. Bowers, "CycloneNTT: An NTT/FFT Architecture Using Quasi-Streaming of Large Datasets on DDR- and HBM-based FPGA Platforms," *Cryptology ePrint Archive*, 2022.

[8] B. K. Alese, E. Philemon, and S. O. Falaki, "Comparative Analysis of Public-Key Encryption Schemes," *International Journal of Engineering and Technology*, 2012.

[9] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large scale Computing Capabilities," *Spring Joint Computer Conference*, 1967.

[10] S. Angel, A. J. Blumberg, E. Ioannidis, and J. Woods, "Efficient Representation of Numerical Optimization Problems for SNARKs," *USENIX Security Symposium*, 2022.

[11] D. W. Archer, B. d. B. Pigem, D. Bogdanov, M. Craddock, A. Gascon, R. Jansen, M. Jug, K. Laine, R. McLellan, O. Ohrimenko *et al.*, "UN Handbook on Privacy-Preserving Computation Techniques," *arXiv preprint*, 2023.

[12] E. Ben Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized Anonymous Payments from Bitcoin," *IEEE Symposium on Security and Privacy*, 2014.

[13] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again," *Innovations in Theoretical Computer Science Conference*, 2012.

[14] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie, "ConsenSys/gnark: v0.9.0," https://github.com/Consensys/gnark, 2023, Last accessed on 06/03/2024.

[15] V. Buterin *et al.*, "Ethereum White Paper," https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf, 2013, Last accessed on 06/03/2024.

[16] A. Crapé and L. Eeckhout, "A Rigorous Benchmarking and Performance Analysis Methodology for Python Workloads," *IEEE International Symposium on Workload Characterization*, 2020.

[17] V. B. Dang, F. Farahmand, M. Andrzejczak, K. Mohajerani, D. T. Nguyen, and K. Gaj, "Implementation and Benchmarking of Round 2 Candidates in the NIST Post-Quantum Cryptography Standardization Process Using Hardware and Software/Hardware Co-design Approaches," *Cryptology ePrint Archive*, 2020.

[18] EECS750 Advanced Operating Systems Course, Institute for Information Sciences, The University of Kansas, "LLC MPKI calculation," https://www.ittc.ku.edu/~heechul/courses/eecs750/S18/eecs750_hw3.pdf, Last accessed on 08/02/2024.

[19] J. Ernstberger, S. Chaliasos, G. Kadianakis, S. Steinhorst, P. Jovanovic, A. Gervais, B. Livshits, and M. Orrù, "zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs," *Cryptology ePrint Archive*, 2023.

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Transactions on Programming Languages and Systems*, 1987.

[21] A. Fiat and A. Shamir, "How To Prove Yourself: Practical Solutions to Identification and Signature Problems," *Conference on the Theory and Application of Cryptographic Techniques*, 1986.

[22] B. Fisch, J. Bonneau, N. Greco, and J. Benet, "Scaling Proof-of-Replication for Filecoin Mining," https://research.protocol.ai/publications/scaling-proof-of-replication-for-filecoin-mining/fisch2018.pdf, 2018, Last accessed on 06/03/2024.

[23] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge," *Cryptology ePrint Archive*, 2019.

[24] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic Span Programs and Succinct NIZKs without PCPs," *Advances in Cryptology – EUROCRYPT*, 2013.

[25] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," *Annual ACM Symposium on Theory of Computing*, 2009.

[26] O. Goldreich, "Secure Multi-Party Computation," *Manuscript. Preliminary version*, 1998.

[27] S. Goldwasser, "Interactive Proof Systems," *Computational Complexity Theory*, 1989.

[28] S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof-Systems," *Providing sound foundations for cryptography: On the work of shafi goldwasser and silvio micali*, 2019.

[29] E. Gorla and M. Massierer, "Point compression for the trace zero subgroup over a small degree extension field," *Designs, Codes and Cryptography*, 2015.

[30] J. Groth, "On the Size of Pairing-Based Non-interactive Arguments," *Advances in Cryptology – EUROCRYPT*, 2016.

[31] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, 1988.

[32] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight Resource Scaling for Cloud Applications," *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012.

[33] D. Hopwood, S. Bowe, T. Hornby, N. Wilcox *et al.*, "Zcash Protocol Specification," *GitHub*, 2016, Last accessed on 06/03/2024.

[34] Y. E. Housni, "Pairings in Rank-1 Constraint Systems," *International Conference on Applied Cryptography and Network Security*, 2023.

[35] HPCWIKI, "HPC Wiki," https://hpc-wiki.info/hpc/Scaling, 2022, Last accessed on 06/03/2024.

[36] IBM, "Zero-knowledge Proofs project," https://research.ibm.com/projects/zero-knowledge-proofs, 2018, Last accessed on 06/03/2024.

[37] IBM, "Cost of a Data Breach Report," https://www.ibm.com/reports/data-breach, 2023, Last accessed on 06/03/2024.

[38] Intel, "Intel VTune Profiler Performance Analysis Cookbook."

[39] Intel, "LLC MPKI calculation," https://community.intel.com/t5/Analyzers/Calculating-LLC-MPKI/m-p/1571364/highlight/true, Last accessed on 08/02/2024.

[40] A. V. Jamet, L. Alvarez, D. A. Jiménez, and M. Casas, "Characterizing the Impact of Last-level Cache Replacement Policies on Big-data Workloads," *IEEE International Symposium on Workload Characterization*, 2020.

[41] Z. Ji, Z. Zhang, J. Xu, and L. Ju, "Accelerating Multi-Scalar Multiplication for Efficient Zero Knowledge Proofs with Multi-GPU Systems," *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

[42] W. Jung, E. Lee, S. Kim, N. Kim, K. Lee, C. Min, J. H. Cheon, and J. H. Ahn, "Accelerating Fully Homomorphic Encryption Through Microarchitecture-Aware Analysis and Optimization," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2021.

[43] H. Kim and S.-C. Park, "Efficient Hardware Acceleration of Chinese Remainder Theorem for Fully Homomorphic Encryption," *International Conference on Electronics, Information, and Communication*, 2023.

[44] M. Kobelt, M. Sober, and S. Schulte, "A Benchmark for Different Implementations of Zero-Knowledge Proof Systems," *IEEE International Conference on Blockchain*, 2023.

[45] J. Ku, J. Zhang, H. Shan, S. Samudrala, J. Wu, Q. Zheng, Z. Li, J. Rajendran, and Y. Chen, "ModSRAM: Algorithm-Hardware Co-Design for Large Number Modular Multiplication in SRAM," *arXiv preprint*, 2024.

[46] V. Lyubashevsky, N. K. Nguyen, and M. Plançon, "Lattice-Based Zero-Knowledge Proofs and Applications: Shorter, Simpler, and More General," *Annual International Cryptology Conference*, 2022.

[47] W. Ma, Q. Xiong, X. Shi, X. Ma, H. Jin, H. Kuang, M. Gao, Y. Zhang, H. Shen, and W. Hu, "GZKP: A GPU Accelerated Zero-Knowledge Proof System," *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2023.

[48] Microsoft, "Advancing Privacy with Zero-Knowledge Proof Credentials," https://techcommunity.microsoft.com/t5/security-compliance-and-identity/advancing-privacy-with-zero-knowledge-proof-credentials/ba-p/1441554, 2020, Last accessed on 06/03/2024.

[49] J. Mo, J. Gopinath, and B. Reagen, "HAAC: A Hardware-Software Co-Design to Accelerate Garbled Circuits," *International Symposium on Computer Architecture*, 2023.

[50] J. Mu, H. Tan, J. Wu, H. Lu, C.-H. Chang, S. Chen, S. Liang, J. Ye, H. Li, and X. Li, "Energy-efficient NTT Design with One-bank SRAM and 2-D

PE Array," *Design, Automation Test in Europe Conference Exhibition*, 2023.

[51] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques," *IEEE Transactions on Computers*, 1999.

[52] SurfShark, "Number of User Accounts Exposed Worldwide," https://www.statista.com/statistics/1307426/number-of-data-breaches-worldwide/, 2024, Last accessed on 06/03/2024.

[53] C. Wang and M. Gao, "SAM: A Scalable Accelerator for Number Theoretic Transform Using Multi-Dimensional Decomposition," *IEEE/ACM International Conference on Computer Aided Design*, 2023.

[54] S.-T. Wang, H. Xu, A. Mamandipoor, R. Mahapatra, B. H. Ahn, S. Ghodrati, K. Kailas, M. Alian, and H. Esmaeilzadeh, "Data Motion Acceleration: Chaining Cross-Domain Multi Accelerators," *IEEE International Symposium on High-Performance Computer Architecture*, 2024.

[55] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A Distributed Zero Knowledge Proof System," *USENIX Security Symposium*, 2018.

[56] C. F. Xavier, "PipeMSM: Hardware Acceleration for Multi-Scalar Multiplication," *Cryptology ePrint Archive, Paper 2022/999*, 2022.

[57] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," *IEEE International Symposium on Performance Analysis of Systems and Software*, 2014.

[58] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-Dive Analysis of the Data Analytics Workload in CloudSuite," *IEEE International Symposium on Workload Characterization*, 2014.

[59] Y. Zhang, S. Wang, X. Zhang, J. Dong, X. Mao, F. Long, C. Wang, D. Zhou, M. Gao, and G. Sun, "PipeZK: Accelerating Zero-Knowledge Proof with a Pipelined Architecture," *Annual International Symposium on Computer Architecture*, 2021.

[60] zk Collective, "zk-harness," https://github.com/zkCollective/zk-Harness, 2023, Last accessed on 06/03/2024.