zk-Bench: A Toolset for Comparative Evaluation and Performance Benchmarking of SNARKs

JENS ERNSTBERGER, Technical University of Munich, Germany
STEFANOS CHALIASOS, Imperial College London, United Kingdom
GEORGE KADIANAKIS, Ethereum Foundation, Greece
SEBASTIAN STEINHORST, Technical University of Munich, Germany
PHILIPP JOVANOVIC, University College London, United Kingdom
ARTHUR GERVAIS, University College London, United Kingdom
BENJAMIN LIVSHITS, Imperial College London, United Kingdom
MICHELE ORRÙ, Centre National de la Recherche Scientifique, France

Zero-Knowledge Proofs (ZKPs), especially Succinct Non-interactive ARguments of Knowledge (SNARKs), have garnered significant attention in modern cryptographic applications. Given the multitude of emerging tools and libraries, assessing their strengths and weaknesses is nuanced and time-consuming. Often, claimed results are generated in isolation, and omissions in details render them irreproducible. The lack of comprehensive benchmarks, guidelines, and support frameworks to navigate the ZKP landscape effectively is a major barrier in the development of ZKP applications.

In response to this need, we introduce zk-Bench, the first benchmarking framework and estimator tool designed for performance evaluation of public-key cryptography, with a specific focus on practical assessment of general-purpose ZKP systems. To simplify navigating the complex set of metrics and qualitative properties, we offer a comprehensive open-source evaluation platform, which enables the rigorous dissection and analysis of tools for ZKP development to uncover their trade-offs throughout the entire development stack; from low-level arithmetic libraries, to high-level tools for SNARK development.

Using zk-Bench, we *(i)* collect data across 13 different elliptic curves implemented across 9 libraries, *(ii)* evaluate 5 tools for ZKP development and *(iii)* provide a tool for estimating cryptographic protocols, instantiated for the $\mathcal{PIOK}$ proof system, achieving an accuracy of $6-32\%$ for ZKP circuits with up to millions of gates. By evaluating zk-Bench for various hardware configurations, we find that certain tools for ZKP development favor compute-optimized hardware, while others benefit from memory-optimized hardware. We observed performance enhancements of up to 40% for memory-optimized configurations and 50% for compute-optimized configurations, contingent on the specific ZKP development tool utilized.

1 INTRODUCTION

Cryptographic proofs play a fundamental role in complexity theory and cryptography. They enable an untrusted prover to convince a verifier that a certain computation was executed correctly. In addition, the provided proof can be *zero-knowledge* [35], indicating that the proof reveals no additional information to the verifier beyond the truthfulness of the proven statement.

In the past decade, general-purpose *Zero-Knowledge Proofs (ZKPs)* [19, 22, 35, 36, 48] transitioned from theory to practice, and are nowadays a core component of many trust-minimized systems. For example, ZKPs are commonly used to prove the integrity of storage [54], to enhance the privacy guarantees of digital asset transfers [20, 21, 31, 60], and to ensure the scalability of blockchain infrastructure [53, 65, 67].

Tools and libraries for ZKPs are typically constructed in multiple layers, encompassing implementations of finite fields, elliptic curves, specific models of computation (such as Rank-One Constraint Systems (R1CS) [9], Quadratic Arithmetic Programs (QAP) [55], Plonkish, and more), polynomial and vector algebra, public-key cryptography, and the actual proof system. Together, these components provide diverse functionality and performance trade-offs, and may further be customized or optimized. Anyone seeking to utilize ZKPs must systematically evaluate the available primitives to determine the most suitable combination for their specific task, taking into consideration the

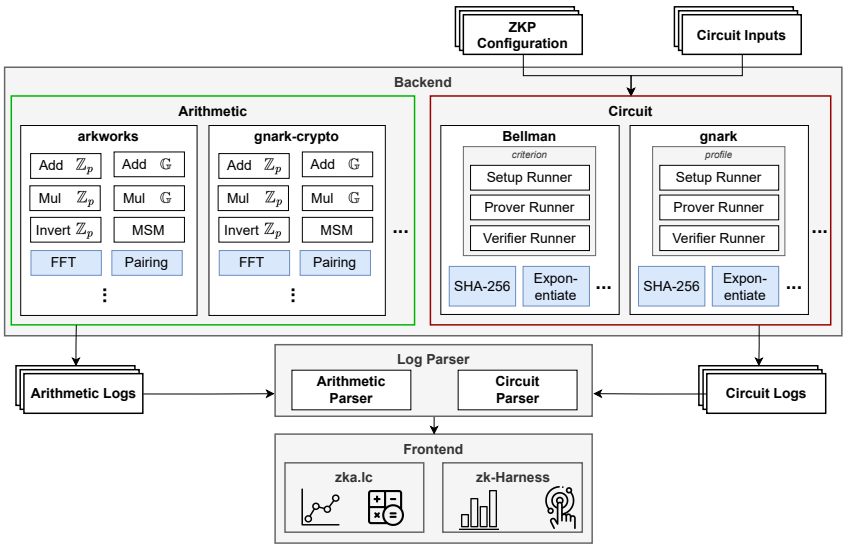


Fig. 1. zk-Bench architecture consisting of three main parts: an *arithmetic backend*, a *circuit backend* and a *dual frontend*. The arithmetic backend covers arithmetic in finite fields and operations on elliptic curves. Whenever applicable, upstream benchmarks are imported. Each ZKP tool or library is benchmarked on equivalent test vectors for comparability. Standardized logs are the base for runtime estimation and the web interface.

nuanced trade-offs inherent to each option. This, however, is a task that is both demanding in terms of expertise, labor and computational resources. Similarly, consulting scientific publications or the broader community frequently proves insufficient. Many cryptographic research endeavors tend to overlook detailed implementations and direct performance assessments, focusing instead on specific elements of a given cryptographic scheme [3, 14, 16, 26]. Consequently, it is infeasible for developers to gain a holistic overview of performance trade-offs, leaving them reliant on fragmented insights and potentially compromising the efficiency and security of their chosen implementations.

Our contributions. In light of these challenges, we introduce zk-Bench, the first open-source, generic, extensible, and configurable benchmarking framework for public-key cryptography and ZKPs. zk-Bench (summarized in Fig. 1) provides extensible backends to benchmark *(i)* low-level arithmetic operations and *(ii)* high-level ZKP circuits with exemplary test vectors, such as an implementation of SHA-256 in a given ZKP tool or library. For high-level ZKP circuits, zk-Bench conducts exhaustive benchmarking, encompassing both execution time and memory consumption for ZKPs for the setup, proving, and verification phases. In addition to providing a set of holistic benchmarks, zk-Bench proposes a novel methodology with which developers and researchers can conduct fine-grained performance analysis of cryptographic algorithms and protocols, as it allows estimating their run-time without actually implementing them. This is not only useful for making informed decisions when choosing the most suitable cryptographic building blocks for a given task, but also for new cryptographic protocols or reproducing scientific results.

To demonstrate the usefulness of zk-Bench, we include 13 different elliptic curves implemented across 9 libraries for arithmetic operations in finite fields, and 5 tools for developing ZKPs. We find that different tools for ZKP development use different approaches for implementing the same ZKP, and present ways in which different open-source implementations can be improved based on the

insights gathered from both the arithmetic backend and circuit backend. By using the benchmarks obtained through the circuit backend as a sanity check, we show that the arithmetic benchmarks can be used to *(i)* accurately extrapolate the performances of complex arithmetic operations and *(ii)* approximate the runtime of a ZKP circuit implementation — without actually implementing it.

To summarize, we make the following contributions:

(1) zk-Bench, the first holistic and extensible benchmarking framework for ZKP development tools and arithmetic libraries.
(2) Comprehensive benchmarks across 9 arithmetic libraries and 5 ZKP frameworks.
(3) A novel methodology for estimating the computational costs of cryptographic protocols, accessible through a web interface.

Benchmark Results. By instantiating zk-Bench with exemplary libraries for ZKP development, we obtain numerous surprising results. Subsequently, we provide a succinct summary of our findings. We suggest that readers unfamiliar with the context first consult Section 2.

For the arithmetic backend, we find that addition and multiplication in the scalar field Z_p of BN254 is not consistently faster than addition and multiplication in the scalar field Z_p of BLS12-381. This is noteworthy, especially since BN254 functions over a field with a smaller characteristic compared to BLS12-381. For benchmarks of elliptic curve operations, we, perhaps counterintuitively, find that `gnark-crypto` outperforms `blstrs` for BLS12-381. For an MSM instance of size 2^{20}, `gnark-crypto` (0.35 s) is 1.83× faster than `blstrs` (0.64s ± 0.03s).

On the circuits side, we observed that the efficiency of ZKPs benefits from optimized low-level arithmetic libraries. For example, an MSM instance of size 2^{20} in `gnark-crypto` is 9.44× faster in G1 and 13.06× faster in G2 when compared to `ffjavascript`, leading to `gnark` being 14.02× faster than `snarkjs` for proving SHA256 with a pre-image size of 16 kB. Another key observation is that custom circuit implementations can falsify the perceived performance of a library or DSL. `halo2`, for example, is 8.64× slower than `gnark` for our first test vector, but due to circuit optimizations, 1.09× faster for the second. When it comes to proof size, we observe that, as expected, `starky`'s proof size is disproportionately larger, being 299.68× to 678.98× greater than the largest Groth16 and $\mathcal{PlonK}$ proofs for $\approx 2^{24}$ constraints/rows. Lastly, we observed that some libraries benefit greatly from compute-optimized hardware, whereas others are more susceptible to memory-optimized hardware, up to 50% and 40% performance increase for proving, respectively. Hence, developers should strategically select their infrastructure depending on the library or tool they utilize.

Paper Structure. This paper is structured as follows: Section 2 presents the relevant background on public key cryptography, ZKPs and implementations of ZKPs in practice. Section 3 introduces zk-Bench, its design decisions, components, and methodology for benchmarking. Section 4 motivates the selection of arithmetic libraries, curves, and tools for ZKP development integrated in zk-Bench. Section 5 presents results and insights as obtained with zk-Bench for the previously selected arithmetic libraries, curves, and tools for ZKP development. In Section 6 we utilize the obtained benchmarks to extrapolate and estimate the runtime of cryptographic protocols. Section 7 discusses the implications of the results obtained through our work, and Section 8 presents the related work. Finally, in Section 9, we conclude the paper.

2 BACKGROUND AND MOTIVATION

2.1 Public-Key Cryptography

Public-key cryptography lies at the foundation of secure and trusted communication [25]. It is widely used in various applications, like secure messaging, digital document signing, web browsing, and monetary transactions. Roughly speaking, tools in public-key cryptography consider groups $\mathsf{G} = \langle G \rangle$ of some prime order p, for which the discrete logarithm (DL) problem is hard: given X

sampled uniformly at random from G, it is hard to find $x \in \mathsf{Z}_p$ (where $\mathsf{Z}_p = \mathsf{Z}/p\mathsf{Z}$ is the field of p elements) such that $xG = X$ (we treat all groups using additive notation). In contemporary cryptography, data protection problems are often formulated using algebraic properties over G and Z_p, where system security relies on computationally intractable problems, such as the DL problem. In practice, we often consider elliptic curve groups over prime fields of large characteristic (as natural in many cryptographic applications), that is, G is the set of points $(x, y) \in \mathsf{Z}_q^2$ solving the equation $y^2 = x^3 + ax + b$ for some fixed parameters $a, b \in \mathsf{Z}_q$.[1]

Operations over the scalar field Z_p and base field Z_q play a crucial role in various aspects of (DL-based) public-key cryptography. These operations introduce a baseline overhead on the top of native CPU instructions, and are integral to the manipulation of polynomials and other algebraic structures, which form the basis of proof systems.

Secondly, the choice of elliptic curve influences functionality, performance, and security of a zero-knowledge proof. Some ZKPs require elliptic curves equipped with a *pairing map*, that is, a non-trivial bilinear map $e : \mathsf{G}_1 \times \mathsf{G}_2 \to \mathsf{G}_T$ mapping two elliptic curve points (typically, two different elliptic curve groups) to some *target group* (typically, a multiplicative subgroup of a finite field) [33]. The estimated security level for *pairing-friendly elliptic curves* is lower than for curves without pairings [50], which would require increasing the size of the coordinate field Z_q. Furthermore, given that novel attacks are introduced frequently, the security of specific curves may degrade over time, requiring again an increase in the size of the coordinate field Z_q. For example, the two curves studied in later sections of this work, BN254 [6, 56] and BLS12-381 [5], were both aiming at a security level of 128 bits when first introduced. However, recent attacks [40, 49] reduce the security level of BN254 to around $100-110$ bits. Successively, the recommendation was to use BLS12 or BN curves over a coordinate field of size 384-bit, instead of BN curves over 256-bit field. The ZCash [37] team followed this recommendation when developing BLS12-381 [18]. Recent work [4], however, suggests that the actual security level of BLS12-381 is only $117 - 120$ bits, instigating recommendations for BLS12 curves over a coordinate field of $440 - 448$ bits[2].

2.2 Zero-Knowledge Proofs (ZKPs)

A zero-knowledge proof is a protocol between a *prover* and a *verifier*, in which the prover is trying to convince the verifier that a certain statement is true without revealing any further information beyond the statement's truthfulness. More formally, the prover attempts to convince the verifier that an *instance* x and a *witness* w are in an NP relation $\mathcal{R}$, i.e., for which $(x, w) \in \mathcal{R}$ can be verified by a Turing machine in polynomial time. A ZKP consists of the following three algorithms:

- $\mathsf{Setup}(\mathsf{pp}) \to (\mathsf{pk}, \mathsf{vk})$. Given public parameters pp as input, compute and output proving and verification keys pk and vk, respectively.
- $\mathsf{Prove}(\mathsf{pk}, x, w) \to \pi$. Given the proving key pk, the instance x, and the witness w, such that $(x, w) \in \mathcal{R}$, as input, compute and output a proof π.
- $\mathsf{Verify}(\mathsf{vk}, x, \pi) \to 0/1$. Given the verification key vk, the instance x, and the proof π as input, output 1 if the proof is valid and 0 otherwise.

As an example, consider a public-key $X \in \mathsf{G}$ and the NP relation $\mathcal{R} = \{(X, x) \in \mathsf{G} \times \mathsf{Z}_p : X = xG\}$. The relation is hard to solve if the DL problem is hard in G, and a zero-knowledge proof for the above relation can convince a verifier that the prover indeed holds a secret-key associated to X, without revealing any information about it. More involved zero-knowledge proofs are used to show satisfiability, i.e., knowledge of a value w for which $C(x, w) = 1$ for some circuit C and input x.

[1]The set of solutions forms an additive group, and for cryptographic purposes we generally focus on a subgroup of prime order p. For more information, we direct the curios reader towards [66].

[2]https://members.loria.fr/AGuillevic/pairing-friendly-curves/

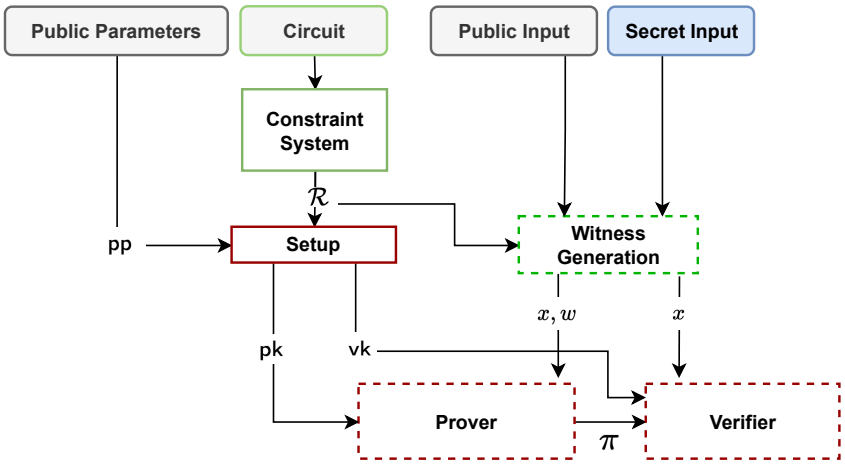


Fig. 2. Practical Implementation of a pre-processing SNARK. The frontend is highlighted in green, the backend is highlighted in red. Public inputs are shaded in grey whereas secret inputs are shaded in blue. Offline operations are represented with solid boxes, whereas online operations are represented with dotted boxes.

Our benchmarks focus on Zero-Knowledge Succinct Non-interactive Argument of Knowledge (SNARK) proofs, where SNARK stands for Succinct (i.e., efficient verification) Non-interactive, ARguments of Knowledge (i.e., proofs where strong soundness properies can be derived from cryptographic assumptions).[3] This is due to their relevance in practice. In particular, we will focus on *pre-processing SNARKs* [22], which introduces another algorithm, called *setup*, that needs to be run only once in an "offline" phase, and allows to pre-process the relation $\mathcal{R}$ which reduces the computational load of the verifier.

2.3 Practical Implementation of Pre-processing SNARKs

We depict the practical implementation of a pre-processing SNARK in Fig. 2. The practical implementation of a pre-processing SNARK involves a two-phase approach. The first phase is offline and preprocesses the computation to a succinct description, while the second phase is online and leverages a SNARK to prove a satisfying circuit assignment to a verifier. The first process of defining the circuit, compiling the circuit to a constraint system, and generating a witness given the assignment of public and secret inputs is commonly referred to as the *frontend* of a SNARK implementation. The *backend*, on the other hand, handles the setup procedure, the generation of the proof as well as the verification of the proof. We highlight each of the steps below.

Backend. The backend has the following components:

- *Setup.* The setup algorithm takes public parameters and a relation $\mathcal{R}$ as input, and outputs the prover key *pk* and verifier key *vk*. We deliberately employ ambiguous terminology — public parameters denote parameters that are independent of the relation being proven. In pre-processing SNARKs this corresponds to the CRS generation algorithm, alternatively termed as *ceremony* [51], or *generator* [22]. For SNARKs in the random oracle model, this corresponds to the selection of the cryptographic hash function. By "setup algorithm", we

[3]Cryptographic assumptions are not sufficient for non-interactive ZKP and SNARKs: additional assumptions (such as the random oracle model, the common reference string model, etc.) are required. In the case of SNARKs, we must also rely on so-called non-falsifiable assumptions [34]. Under these additional constraints, SNARKs exist for any NP statement [13].

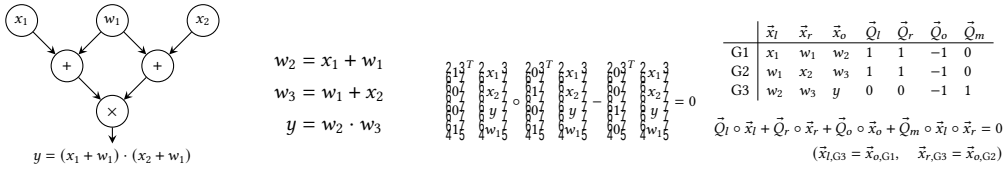


Fig. 3. The arithmetic circuit $C(x_1, x_2, w_1) = (x_1 + w_1) \cdot (x_2 + w_1)$ over Z_q in different representations. (a) Graph representation. (b) Constraints for the depicted arithmetic circuit. (c) R1CS representation for the circuit ("$\circ$" denotes the Hadamard product). (d) Representation of the arithmetic circuit as a "Plonkish" table.

refer to the relation-specific setup. The relation-specific setup is also frequently referred to as *indexer* in preprocessing SNARKs [15, 22, 32]. Note that the indexer is an offline, public-coin algorithm, it does not depend on any witness or specific instance assigned to $\mathcal{R}$.

- *Prover & Verifier.* The prover and verifier collaboratively run the online phase of the preprocessing SNARK. The prover computes a proof convincing the verifier that an assignment of private and public inputs to the circuit satisfies $\mathcal{R}$. To do so, the prover leverages pk in the proof computation, and the verifier leverages vk in the verification.

Frontend. The frontend has the following components:

- *Circuit.* A computation is represented as an arithmetic circuit in the frontend of a SNARK. An arithmetic circuit is a directed acyclic graph, where the edges are denounced *wires* and the vertices are called *gates*. An arithmetic circuit supports addition and multiplication gates, with unrestricted fan-in and unrestricted fan-out. We depict an exemplary arithmetic circuit with fan-in 2 and fan-out 1 in Fig. 3. Additional tooling allows developers to specify the circuit through a domain-specific programming language or a SNARK library.
- *Constraint System.* A tool for development of SNARKs takes the circuit specification as an input, and compiles it to an *intermediate representation*. At the time of writing, the most popular intermediate representations are Rank-1 Constraint System (R1CS) [55], Arithmetic Intermediate Representation (AIR) [8] and Plonkish [32].
- *Witness Generation.* To generate a transcript of wire assignments to the arithmetic circuit, the intermediate representation is assigned with both the public and private inputs of the statement to be proven. In some tools for SNARK development, the process of witness generation is not a distinct step but rather integrated into the prover API. For clarity and ease of understanding, we separate out the witness generation in our explanation.

3 ARCHITECTURE

In this section, we present the architecture of our framework: the core design decisions, the structure of the system, and the methodology used for collecting samples and estimating performance.

3.1 Design Decisions

Goals. The primary goal of zk-Bench is to provide a valid, fair, and methodologically sound system to measure performance. To achieve this goal, we pursue the following set of points:

G1 Completeness: zk-Bench covers quantitative metrics for components shown in Fig. 2, including low-level arithmetic operations, public-key operations, proof system performance, and circuit execution. As of now, we support 9 arithmetic libraries covering the implementation

of 13 elliptic curves and 5 tools for ZKP development. Our framework accounts for ∼ 3 300 lines of Go, ∼ 2 600 lines of Rust, ∼ 2 000 lines of Python, and ∼ 1 800 lines of JavaScript code.

G2 Modularity: Whenever possible, we leverage benchmarks from the upstream library, enhance them as required, and contribute our modifications back to the original source. We rely on popular benchmarking libraries to perform accurate timing measurements. Our framework design is modular to facilitate extension by external developers: each arithmetic operation and circuit is tested separately, and the arithmetic benchmarks are isolated from circuit benchmarks. Additionally, little developer effort is required to extend our circuit benchmarks. For instance, integrating Bellman into zk-Bench necessitates 81 lines of code (LOC) in Rust to automate the benchmarking of circuits, along with 40 lines of Python code to add it to our framework. In contrast, adding a new circuit requires an additional 30 lines of Rust code for benchmarking registration. For a DSL such as Circom, integration into zk-Bench demands more effort, with 210 lines of scripting code plus 40 lines of Python code, whereas adding additional circuits needs no extra LOC beyond the circuit itself.

G3 Reproducibility: Benchmarks are automated and executed on standardized cloud hardware with reproducible configurations. This ensures consistent and comparable results, facilitating reliable performance evaluations. Benchmarking results are publicly accessible.

G4 Pragmatism: Our objective is to evaluate libraries under conditions that mirror practical deployments: we set optimization flags whenever possible, and use all cores available in the machine. Such a setup mirrors the common usage of these tools in real-world applications and does not compromise the accuracy of the measurements (cf. Section 5). To the best of our knowledge, there are no standard formats for recording raw benchmark data. We develop a unified format that can be used in the future also by other developers.

Metrics. In a zero-knowledge proof system, the main performance evaluation metrics are computation time, computation space, and proof size. We capture all the above in the following way:

M1 Execution time: We rely on the following state-of-the-art libraries and profilers: (a) for Rust, we selected criterion.rs [12]; (b) in Go, we selected the standard library's benchmark toolsuite testing package for the arithmetic backend, and a CPU profiler for the circuit backend; (c) for JavaScript, we rely on tinybench [1]. All libraries adaptively select the number of samples, and return the average running time[4].

M2 Memory consumption: Due to the absence of a precise universal memory measurement framework and varying footprints across programming languages, we limited our comparison to memory usage within tools for developing general-purpose ZKPs. zk-Bench generates binaries that are independent of the ones used for evaluating execution time, and measures the *maximum resident set size* throughout its execution. We include the serialization/deserialization of relevant values in each phase to measure memory consumption of the setup algorithm, prove and verify in isolation and accurately reflect real-world usage of proof systems.

M3 Proof size: We consider the byte-size of a serialized proof. Since proofs are cryptographic objects, the proof size cannot be shrunk significantly using compression algorithms.

3.2 Backend

Two core components constitute our framework:

Arithmetics Backend. The arithmetics backend is responsible for benchmarking low-level arithmetic operations of supported libraries, e.g., operations over finite fields and elliptic curves. We benchmark the following operations:

[4]Another possible approach here would be to instead return the mode. Conformingly to our requirement of modularity, we opted for leaving the default metric of those libraries unchanged.

- **Field operations**: for $a, b \in \mathsf{Z}_p$ sampled uniformly at random, we measure addition $a + b$, multiplication $a \cdot b$, inversion a^{-1}.
- **Elliptic curve group operations**: for $a \in \mathsf{Z}_p$ and $P, Q \in \mathsf{G}$ sampled uniformly at random, we measure group addition $P + Q$ and scalar multiplication aP (a so-called Diffie-Hellman). We do not concern ourselves with different encodings and algorithms behind elliptic curve operation. Instead, we measure basic APIs offered by each library, as they represent the primary interfaces that most library users will engage with.
- **Elliptic curve pairing operations**: For elliptic curves equipped with a pairing map $e : \mathsf{G}_1 \times \mathsf{G}_2 \to \mathsf{G}_T$, we measure the pairing time $e(A, B)$ for A, B uniformly random in $\mathsf{G}_1, \mathsf{G}_2$.
- **Amortized operations**: Our focus extends to cryptographic operations that benefit from amortization using tailored algorithms, prompting us to assess the amortized runtime. Specifically, we examine computation of an inner-product in the field: for some $n \; ¡ \; 0$, given $\vec{a}, \vec{b} \in \mathsf{Z}_p^n$, compute $\prod_i a_i b_i$ [47]; multi-scalar multiplications: given $\vec{a}, \vec{G}$, compute $\prod_i a_i G_i$ [57]; multi-pairing: given $\vec{A} \in \mathsf{G}_1, \vec{B} \in \mathsf{G}_2$, compute $\prod_i e(A_i, B_i)$; the number-theoretic transform (NTT) in fields with high 2-adicity [24].

The number of samples for each operation is decided adaptively by the underlying benchmarking library. Amortized operations are sampled across the geometric sequence of sizes $2^1, 2^2, \ldots, 2^{22}$ and more. For each sample, we make sure that at least 10 executions are performed. Note, that there are various approaches to perform multi-scalar multiplication with individual trade-offs [11]. In zk-Bench, we do not differentiate between the specific optimizations chosen in individual implementations. Instead, we benchmark the given libraries as is, without altering existing methods.

Circuit Backend. For every supported tool for ZKP development, the circuit backend includes three distinct runners, each designated for setup, proving, and verifying phases. The configuration file describes the system under test. It specifies the library/DSL to benchmark, the algebraic structure (that is, finite field or elliptic curve) to rely on, the backend, and the inputs to the circuit.

To precisely measure the execution time, we adopt the same micro-benchmarking libraries, making sure that a minimum of 10 samples are gathered for larger circuits. Upon successfully benchmarking the circuit, the execution time is stored in publicly accessible CSV files. Arbitrary payloads can be added with ease. The benchmark runners can remain untouched, whereas the developer only provides a circuit implementation to evaluate its performance.

3.3 Log Parser

We provide two specialized parsers that convert micro-benchmarking tool outputs into standard formats, probing for the different arithmetic operations and elliptic curves being tested. The processed results, available in both JSON and CSV, are utilized by the supplied front-ends.

3.4 Frontend

We design two frontends to help visualize and extract information from the collected benchmarks.

- **Zkalc**: Zkalc is a JavaScript library and public website using the benchmark data to present *(1)* estimates for the execution times of cryptographic operations, and *(2)* graphs to compare performance across different libraries and programming languages;
- **zk-Harness**: zk-Harness is a web application that leverages the benchmark data to illustrate the computation time, memory, and proof size for each ZKP development tool or library, across various payloads. It enables comparisons between different implementations and configurations.

Table 1. (a) Overview of arithmetic libraries and elliptic curves currently present in our benchmarking framework. (b) Overview of ZKP libraries and proof systems as included in our benchmarking framework, and curves/fields that we consider in the results section. *We use Circom with Snarkjs and Rapidsnark.

(a)

	BLS12-381	BN254	curve25519	BLS12-377	pallas	vesta	secp256k1	jubjub	BW6 curves
curve25519-dalek (4.1.1)			●						
pasta_curves (0.5.1)					●	●			
gnark_crypto (0.11.1)	●	●		●				●	●
arkwork-rs (0.4.2)	●	●		●					
ffjavascript (0.3.59)		●							
halo2_curves (0.3.1)		●			●	●	●		
pairing_ce (0.28.5)	●	●							
zkcrypto (0.23)	●	●							
blstrs (0.7.0)	●								

(b)

		Bellman [69]	PSE/Halo2 [28]	Gnark [17]	Circom [39]*	Starky [59]
Curve/ Field	BLS12-381	●		●	●	
	BN254	●	●	●	●	
	Goldilocks					●
Backend	Groth16	●		●	●	
	Halo2/Plonk (KZG)		●			
	Stark					●

While the former is targeted for generic public-key operations and cost estimations (which we believe can be useful also outside the zero-knowledge space, such as identity- and attribute-based encryption, functional encryption, etc.), the latter is targeted at zero-knowledge circuit developers.

4 SELECTED ARITHMETIC LIBRARIES AND TOOLS FOR ZKP DEVELOPMENT

We implement benchmarks for 13 different elliptic curves across 9 different libraries in Rust, Go, and JavaScript (Wasm) in the arithmetics backend, and 5 tools that provide functionalities to build ZKP applications in the circuit backend.

When benchmarking SNARKs, we focus on BN254[5] and BLS12-381, two extensively used curves, especially in relation to SNARKs, and implemented in the selected libraries and tools we benchmark (cf. Table 1b). For STARKs, we operate in the Goldilocks field, as it is the only one used by `starky`.

In this section, we outline the reasoning behind our selection. Note that zk-Bench is designed for modularity and ease of extension — moving forward, we aim to extend our holistic comparison by supporting further arithmetic libraries and ZKP development tools.

4.1 Arithmetic Libraries

We opted to support a diverse range of arithmetic libraries for executing generic public-key operations. Our goal is not just to understand the differences in their implementations and curves, but also to discern their potential impacts on applications and libraries that further rely on them.

Firstly, we selected arithmetic libraries that form the core of popular and state-of-the-art tools for developing ZKPs. Such libraries include arkworks-rs (Arkworks [2]), pasta_curves (Halo2 [68]), halo2_curves (PSE/Halo2 [28]), zkcrypto (Bellman [69]), pairing_ce (Bellman-ce [42]), ffjavascript (Snarkjs [38]), and gnark_crypto (Gnark [17]). Further, we included blstrs as a baseline for curve BLS12-381, a library extensively utilized by various Ethereum projects, and curve25519-dalek, a widely adopted library used by more than 70,000 repositories. Our selection of these 9 libraries encompasses a wide range of fields, curves, and state-of-the-art optimizations. The details of the selected arithmetic libraries are summarized in Table 1a.

[5]Note: different libraries sometimes refer to the same curve using different names. BN254 is also called BN256 and alt_bn128.

Table 2. Hardware specification as applied in the experimental infrastructure.

Name	RAM	OS	#vCPU	CPU
m5.large	8 GB	Amazon Linux 23	2	Intel Xeon E5-2686
m5.2xlarge	32 GB	Amazon Linux 23	8	Intel Xeon 8175M
m6i.8xlarge	128 GB	Amazon Linux 23	32	Intel Xeon 8375C
m6g.8xlarge	128 GB	Amazon Linux 23	32	Graviton2
r6i.8xlarge	256 GB	Amazon Linux 23	32	Intel Xeon 8375C
c6i.12xlarge	128 GB	Amazon Linux 23	48	Intel Xeon 8375C

4.2 Tools for ZKP Development

In selecting a variety of initial tools for ZKP development in our experiments, we aim to reflect the current space of systems from a neutral perspective, without subjectively favoring one tool over another. Therefore, we identified popular libraries and DSL by examining their popularity, quantified through their number of Github stars, and maintenance, quantified by their most recent commit (cf. Appendix A). We include `circom`, `gnark` and `bellman` due to their popularity. Note that we do not include libsnark, due to insufficient maintenance in the recent past and a decreasing relevance among practically applied solutions. Beyond popularity and maintenance, we included `starky` and `halo2_ce`. `halo2_ce` is a fork of `halo2` by Ethereum's Privacy-Scaling-Exploration (PSE) that replaces the Inner Product Argument with the Kate, Zaverucha, Goldberg (KZG) polynomial commitment scheme. [6] This decision was largely influenced by *(i)* the requirement of including a Succinct Transparent Argument of Knowledge (STARK) as opposed to a ZKP based on pairing-friendly elliptic curves and *(ii)* the relevance of `starky` in the upcoming Polygon zkEVM and `halo2` in the upcoming zkEVM of Scroll.

The tools for developing ZKPs examined in this study, as detailed in Table 1b, span an extensive range of arithmetizations and methodologies. Four of the tools support pairing-based SNARKs, one accommodates STARKs, while three utilize R1CS. Additionally, two of the tools are built to support Plonkish arithmetization. Moreover, among the selected tools for SNARK development, one offers a DSL, while the remaining provide libraries, with three providing APIs in Rust and one in Golang. This diversity in the supported architectures and programming languages ensures a broad coverage, thus allowing for more comprehensive and insightful results.

5 BENCHMARK RESULTS

5.1 Experimental Infrastructure

All benchmarks were executed with four different hardware setups, summarized in Table 2.

- Setup `Small` (AWS m5.large): This setup represents an entry-level cloud server configuration, suitable for lightweight applications.
- Setup `Personal` (AWS m5.2xlarge): This setup emulates the hardware of a midgrade laptop for personal use and general-purpose computation.
- Setup `Server` x86 (AWS m6i.8xlarge): This setup emulates the hardware of a server for general-purpose computation with a large amount of memory.
- Setup `Server` ARM (AWS m6g.8xlarge): We run zk-Bench on different processors to evaluate the impact of CPU architectures on SNARK performance.
- Setup `Memory Optimized` machine (AWS r6i.8xlarge): This machine provides an enhanced RAM configuration, tailored for tasks requiring significant memory consumption.
- Setup `CPU Optimized` machine (AWS c6i.12xlarge): Designed to prioritize processing power, this setup is ideal for compute-heavy applications and parallel processing tasks.

[6] Note that for the rest of the paper we use `halo2` to refer to PSE's fork.

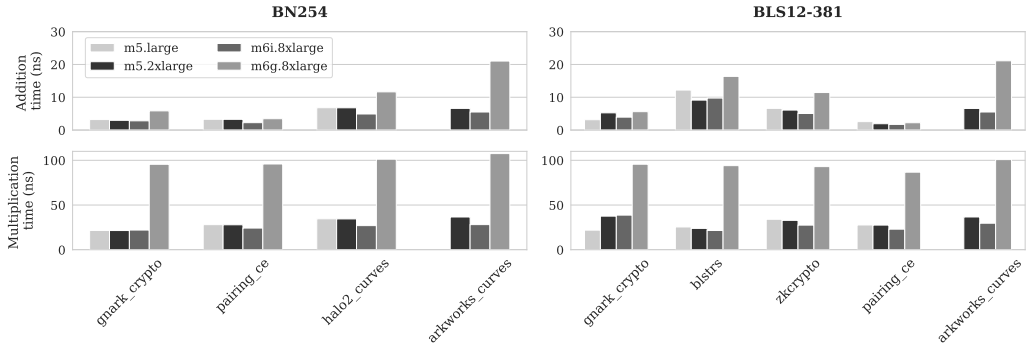


Fig. 4. BN254 and BLS12-381 Field Addition and Multiplication.

5.2 Arithmetics

Figure 4 depicts the micro-benchmarks for addition and multiplication in the scalar field of curve BN254 and BLS12-381 on all machines in our experimental infrastructure; Figure 6 depicts the micro-benchmarks for MSM in G_1 and G_2 for curves BN254 and BLS12-381 on the server sized instance AWS m6i.8xlarge. Although the arithmetic backend supports more than the libraries depicted in Figures 4 and 6, we depict only the selected libraries as they provide the implementation of arithmetic for the most popular SNARK libraries to date (cf. Table 4). In Appendix B, we discuss the validity of our results in more detail.

Field Arithmetic. In Figure 4 we compare the execution time for field addition and field multiplication over the scalar field of the respective curve. We find that:

- At the time of writing, Botrel *et. al* [16] provide the most recent scientific result on optimization for Montgomery multiplication in finite fields, implemented in gnark-crypto; state-of-the-art inner-product of field elements is described by Longa [47]. We find that for BN254, gnark-crypto is on average 30% faster than any other implementation for addition and multiplication in the scalar field Z_p. For BLS12-381, however, gnark-crypto incurs an average slowdown of 9% compared to other libraries.
- The choice of hardware significantly impacts the execution time. Whereas for BLS12-381, the smallest machine in our experimental setup, m5.large, is 13% faster than m5.2xlarge, 5,7% faster than m6i.8xlarge and 71,3% faster than m6g.8xlarge, for BN254 m5.large is 1% slower than m5.2xlarge, 12,4% slower than m6i.8xlarge — but 72,1% faster than field multiplication in m6g.8xlarge when comparing the mean of the execution time across all libraries.
- When it comes to differences between curves, we observe that BN254 is not faster than BLS12-381 across the board, although BN254 operates over a prime field of smaller characteristic than BLS12-381. It can further be observed that the degradation in performance when switching from BLS12-381 with a 381 bit coordinate field to BN254 with a 256 bit coordinate field differs across machines. The loss factor for m5.large, m5.2xlarge, m6i.8xlarge and m6g.8xlarge is 1.18×, 1.21×, 1.15× and 1.09× respectively.
- Field addition is 5,1× to 27,7× faster than field multiplication for BN254 and 2,1× to 38,9× faster than field multiplication for BLS12-381. Specifically, the minimum ratio for BN254 is observed on an AWS m5.large machine with an x86 processor, while the maximum ratio is found for the server size AWS m6g.8xlarge machine with ARM processor. Similarly, for

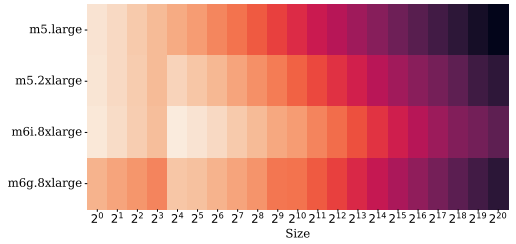
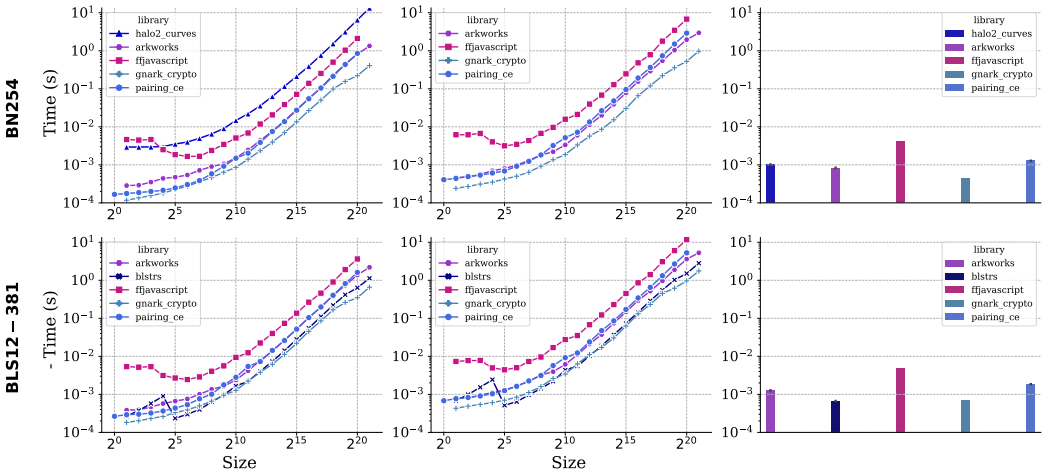


Fig. 5. Multiscalar Multiplication in G_1 of curve BLS12-381 for `blstrs` on all machines as defined in our experimental setup. Values are normalized between 0 and 1, where 1 indicates the highest execution time (darker color) across all machines. Preceding the normalization, we perform a logarithmic transformation to accommodate for the logarithmic increase in the size of MSM instances.



(a) MSM $\mathbb{G}_1$ **(b)** MSM $\mathbb{G}_2$ **(c)** Pairing

Fig. 6. Elliptic Curve Operations for BN254 and BLS12-381 on AWS m6i.8xlarge.

BLS12-381, the minimum ratio is observed on an AWS m5.large machine, while the maximum ratio is observed on the server size AWS m6g.8xlarge machine with ARM processor.

- We observe that `ffjavascript` is nearly two orders of magnitude slower than any other implementation. Specifically, for addition in Z_p of BN254, `ffjavascript` takes 413ns. The reason for this is additional serialization in the *add* function that is exposed by the `ffjavascript` API. [7] The same behavior can be observed for multiplication in Z_p, and also for BLS12-381.

Elliptic Curve Operations. We depict the results of our experiments for elliptic curve operations obtained through zk-Bench in Figures 5 and 6 . Figure 6 depicts elliptic curve operations over BN254 and BLS12-381 on AWS m6i.8xlarge. We make two observations:

- Some implementations use multi-threading only for large batch sizes (8 for ffjavascript, 16 for blstrs). This causes an abnormal spike at the beginning of the curve.

[7] We excluded `ffjavascript` from Fig. 4 for visual clarity.

- For large multi-scalar multiplications, gnark adopts batch affine addition instead of using the projective formulae. This causes the slight downward deviation towards the end of the graph for both gnark and blstrs. For smaller MSM instances, mixed addition in extended Jacobian coordinates delivers an additional performance improvement for gnark and blstrs [16]. A similar approach can similarly improve the efficiency of other libraries.

More concretely, for a MSM instance of size 2^{20} over BN254, `gnark-crypto` takes 0.22s in G_1 and 0.53s in G_2, whereas the second-fastest implementation, provided by `pairing_ce`, takes 0.85s±0.00s in G_1 and 2.91s ± 0.01s in G_2. `gnark-crypto` is 3.86× faster in G_1 and 5.53× faster in G_2. At 0.45ms, the pairing product in `gnark-crypto` is 2.83× to 9.24× faster than `pairing_ce` (1.28ms ± 0.00ms) and `ffjavascript` (4.18ms), respectively. This is expected, as `gnark-crypto` implements the state-of-the-art MSM, a variant of the Pippenger algorithm [27].

For BLS12-381, though, `blstrs` is 1.41× faster for an MSM instance of size 2^5 in G_1. Perhaps counter-intuitively (given the underlying languages used), `gnark-crypto` outperforms `blstrs` for larger size MSM instances and computing the pairing product. For an MSM instance of size 2^{20}, `gnark-crypto` (0.35 s) is 1.83× faster than `blstrs` (0.64s ± 0.03s) in G_1. Similar behavior can be observed in G_2. It should be highlighted that blst utilizes field arithmetic assembly code generalized for fields up to 384-bits, whereas Gnark employs BLS12-381-specific assembly.

To investigate the influence of different machine specifications and instructions set architectures, we depict MSM over BN254 in G_1 in the `blstrs` library for all machines in our experimental setup in Fig. 5. The figure shows a heatmap, where each value is logarithmically transformed, to account for the logarithmic scaling of the MSM size, and successively normalized. The darker the color, the higher the execution time relative to the lowest and highest execution time across all machines. It can be observed, that low-level arithmetic operations are faster for more powerful CPUs (i.e., more CPU cores) in the first three rows. The last row depicts execution on an ARM-based machine, where it can be seen that executing finite field arithmetic is significantly less performant than on x86. We expect a detailed investigation of performance improvements for MSM on the ARM instruction set architecture, an interesting area for future work. Note, that this figure is representative, different libraries besides `blstrs` exhibit a similar pattern across different machines.

5.3 Circuits

In this section, we employ zk-Bench to benchmark two circuits: SHA-256 and Exponentiate, implemented across 5 tools for developing ZKP applications. An overview of the ZKP tools considered can be found in Section 4.2. For each tool, we investigate the time and memory consumption associated with the Setup, Prove, and Verify phase of the underlying SNARK. Our primary target for these benchmarks is AWS m6i.8xlarge, with the results for this server depicted in Fig. 7, 8, and 9. Additionally, given a predefined budget, we examine whether CPU-optimized (c6i.12xlarge) or Memory-optimized (r6i.8xlarge) machines offer better results for individual ZKP libraries and tools.

Test Vectors. The lack of universal test vectors for ZKPs prompted us to propose two initial circuits in our performance evaluation — a circuit proving knowledge of an exponent, and a circuit proving knowledge of a pre-image to a hash computed by the SHA-256 hash function.

- **Exponentiate.** Our benchmark suite includes a finite field exponentiation circuit. This circuit has a clearly defined specification that can be easily represented by most modern proof systems. We manually implemented this circuit consistently across all supported ZKP development tools. The `Exponentiate` circuit takes three variables X, E, Y as public inputs. Then, it computes and constraints $Y' = X^E$, by executing E multiplications, before setting a constraint for $Y' = Y$.

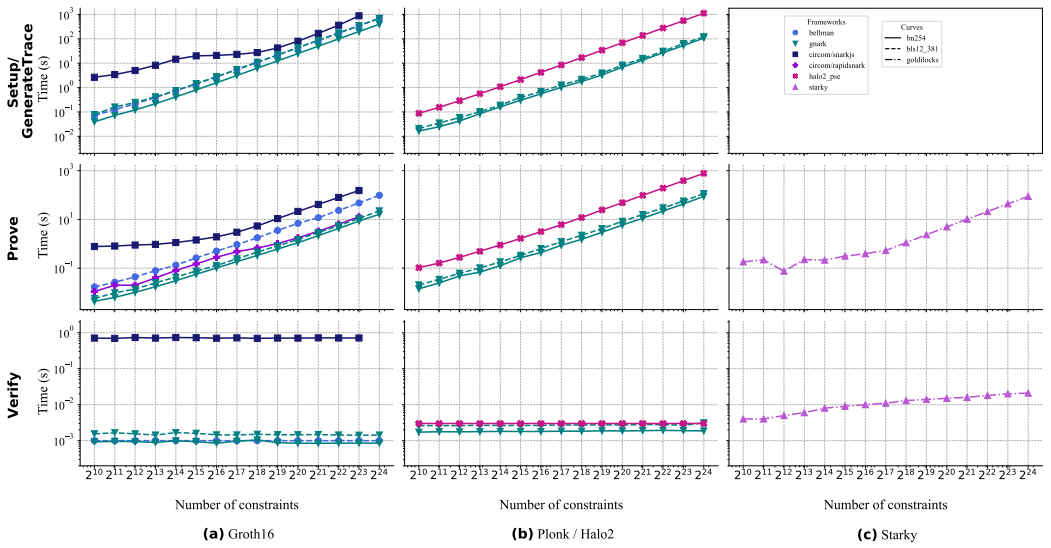


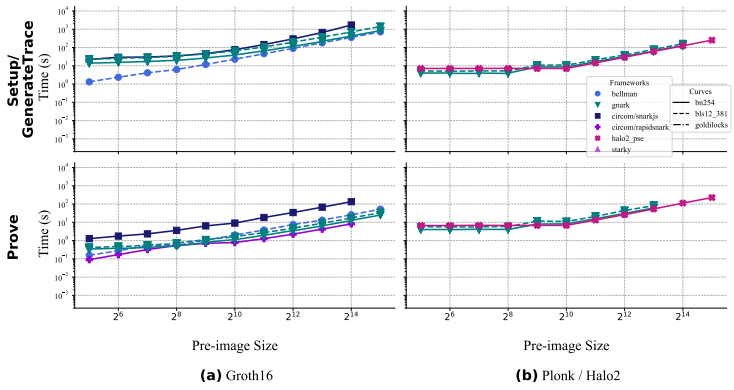
Fig. 7. Execution Time of Exponentiate circuit on m6i.8xlarge. Note that for halo2 and starky instead of constraints, this figure depicts the number of utilized rows.

- **SHA-256.** For the SHA-256 circuit, we rely on implementations as provided in the respective standard libraries of the examined ZKP development tools. We do not consider implementations with insufficient tests, or missing acknowledgment by maintainers. However, it is important to recognize that the performance of a specific circuit implementation may not be representative of the overall proof system's performance, as certain implementations may incorporate more optimizations than others. Therefore, while the SHA256 circuits are valuable for assessing ZKP development tools in a real-world context, absolute conclusions drawn from the results should be approached with caution, as they might not fully encapsulate the inherent capabilities of a given tool.

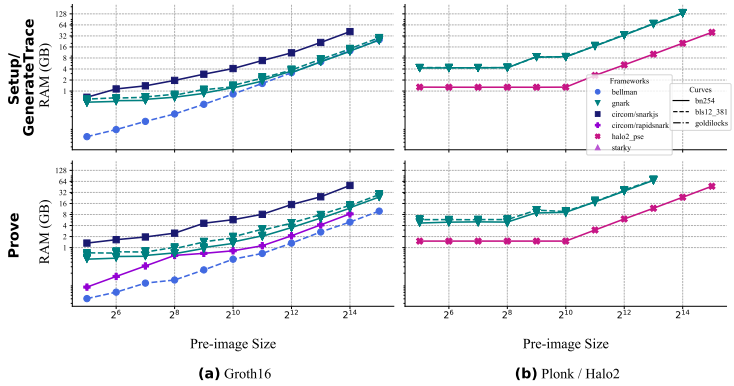
Setup Phase Analysis. The setup is a pre-processing phase performed once per circuit to generate the *pk* for the prover and *vk* for the verifier, respectively (cf. Section 2). Note that starky does not require a setup phase, so it is excluded.

Overall, our results show that the setup phase for the $\mathcal{PLONK}$ proof system is faster than that of Groth16. Specifically, for the SHA-256 circuit, the gnark setup phase with Groth16 and a preimage size of 8kB takes 220.17s. In contrast, the gnark setup phase with $\mathcal{PLONK}$ requires 62.50s. This indicates that the setup for $\mathcal{PLONK}$ is 3.52× faster than of Groth16. A similar trend can be observed for halo2 when being compared to gnark Groth16. However, this efficiency appears to be accompanied by a memory trade-off. gnark's $\mathcal{PLONK}$ implementation consistently consumes *more than twice* the amount of memory compared to its Groth16 counterpart. This increased memory requirement resulted in the inability to successfully generate the proof for preimages of size 16kB and 32kB in the SHA256 circuit due to exceeding the machine's available memory of 128GB.

Notably, in the case of the Exponentiate circuit, the gnark Groth16 setup is faster than the setup in halo2 (e.g., 2.06× faster for 2^{24} multiplication constraints). However, we also observe that the memory footprint of halo2 is consistently lower than the one for gnark, with savings ranging from a minimum of 22.64% to a maximum of 60.56% in the range of 2^{10} to 2^{24} constraints/rows (see Figure 11 in the appendix). Yet, it is important to highlight that, notwithstanding the current



(a) Execution time.



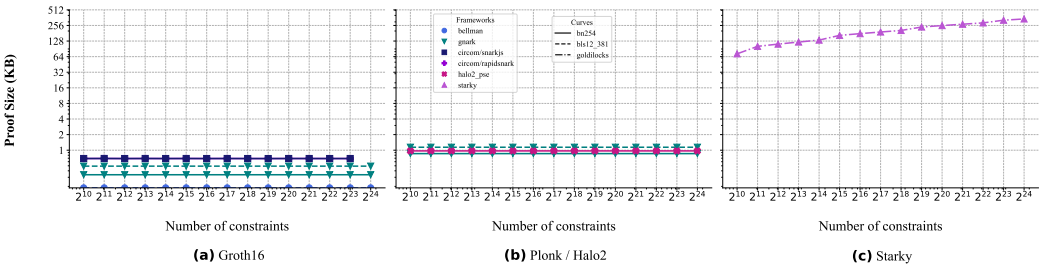
(b) Memory consumption.

Fig. 8. Execution time and memory benchmarks for SHA-256 circuit for preimages ranging from 32 Byte to 32kB on m6i.8xlarge.

execution metrics of the halo2 circuit, optimizations targeting the row count of the exponentiate circuit in halo2 can offer an improve execution time.

Proving Phase Analysis. Contrary to expectations, starky's proving cost for the Exponentiate circuit is 5.00s for 2^{20} multiplications, which is 8.06× slower than gnark $\mathcal{P}\mathfrak{lon}\mathcal{K}$ and merely 3.03× faster than snarkjs, which employs Groth16. One explanation is that starky is optimized for wider traces. Should the circuit be optimized to accommodate a larger number of columns, starky is anticipated to outperform other ZKP development tools in terms of execution time. This is attributed to the fact that rows transform into leaves in a Merkle tree, and smaller rows do not optimize the hash rate in starky. A comprehensive exploration of optimizations is outside the scope of this paper, but could be facilitated by zk-Bench. It's worth noting that similar optimizations can benefit halo2. However, such optimizations might induce higher costs with regard to verification time and proof size.

In contrast to the setup phase, $\mathcal{P}\mathfrak{lon}\mathcal{K}$ trails Groth16 in efficiency. For illustration, within gnark, for SHA256 with a pre-image size of 8 kB, $\mathcal{P}\mathfrak{lon}\mathcal{K}$ is 9.39× slower compared to Groth16. This pattern remains consistent across various benchmarks and also applies to halo2.



(a) Groth16 **(b)** Plonk / Halo2 **(c)** Starky

Fig. 9. Proof Size for the Exponentiate circuit as observed on m6i.8xlarge. The observed proof size for circom/snarkjs and circom/rapidnsark is the same.

In terms of memory consumption, ZKP development tools in Rust exhibit a substantial reduction compared to other tools for ZKP development. For instance, given a 32 kB preimage size, `bellman`'s memory requirement is 10.62 GB. In contrast, `gnark` requires 30.55 GB for Groth16 and exceeds memory limits for $\mathcal{P}\mathfrak{lonK}$, outstripping `bellman` by a factor of 2.88× for Groth16 over BLS12-381. Interestingly, `bellman`, despite not being the fastest, emerges as the most efficient in terms of memory utilization.

Similar to our observations from Section 5.2, the advantage of BN254 over BLS12-381 is evident in proving times. Specifically, the `gnark`-BN254-Groth16 combination consistently outperforms the `gnark`-BLS12-381-Groth16 results.

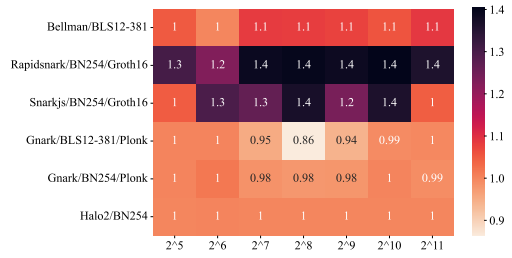
Both `gnark` and `rapidsnark` stand out as the implementations with the most efficient provers. However, `gnark`'s superior performance is offset by its suboptimal memory efficiency. Specifically, for SHA256 with a pre-image size of 16 kB, the time metrics for `gnark` over BN254, `rapidsnark`, `bellman`, and `snarkjs` are 17.10s, 8.24s, 25.81s, and 134.20s, respectively. In terms of memory consumption, `gnark` requires 15.25 GB, in contrast to `rapidsnark`, which utilizes only 8.92 GB.

Veri□cation Phase Analysis. Fig. 7 showcases the verification results across the evaluated tools. Aligned with theoretical expectations, the verification time for Groth16 and $\mathcal{P}\mathfrak{lonK}$ remains both constant and minimal. Conversely, for `starky`, this time scales with input size. To illustrate, while all tools and libraries for SNARK development consistently verify within a 5 ms window, `starky` can require upwards of an order of magnitude more (e.g., 22ms for 2^{25} multiplication constraints).

Proof Size Analysis. Fig. 9 presents the comparative proof sizes for the Exponentiate circuit across various tools for ZKP development. As expected, a standout observation is the exponential difference in `starky`'s proof size—ranging from 299.68× to 678.98× greater for roughly 2^{24} constraints/rows–when compared against the larger Groth16 and $\mathcal{P}\mathfrak{lonK}$ proofs. As anticipated, the proof sizes for Groth16, $\mathcal{P}\mathfrak{lonK}$, and `halo2` remain constant. Note that we have excluded `bellman` from the comparison because its proof is measured in raw bytes, whereas the other proofs are serialized.

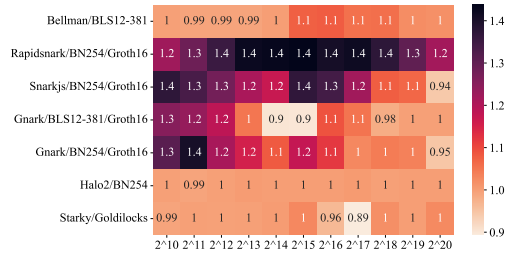
For plonkish arithmetization, an increase in column count corresponds to an elevated proof size. The Exponentiate circuit in `halo2` features a proof size of 992B, whereas the SHA256 circuit, accommodating a greater column count, has a proof size of 2816B. Nevertheless, adding more columns can introduce various optimizations, primarily aimed at reducing row count to enhance proving time.

Memory Optimized vs. CPU Optimized Machines. We set out to determine whether a specific tool demonstrates superior performance on compute-optimized or memory-optimized hardware. To investigate this, we run our experiments on two AWS instances: c6i.12xlarge (48 vCPU, 128 RAM) and r6i.8xlarge (32 vCPU, 256 RAM). Though they differ in specifications, their hourly costs

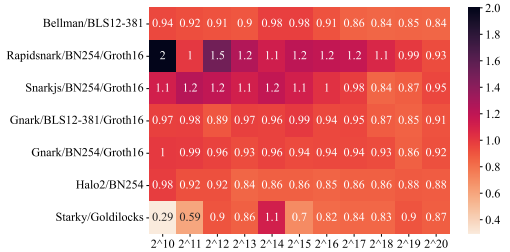


(a) SHA256 - Memory Consumption

(b) SHA256 - Execution Time



(c) Exponentiate - Memory Consumption



(d) Exponentiate - Execution Time

Fig. 10. Performance decrease/increase for SHA256 and Exponentiate circuits in terms of memory consumption and execution time for CPU optimized (c6i.12xlarge) vs. RAM optimized (r6i.8xlarge) machines for *proving*. For execution time, darker color indicates that the CPU-optimized machine requires more time than the RAM-optimized machine (i.e., the RAM-optimized machine outperforms the CPU-optimized machine). For memory consumption, darker color indicates that the CPU-optimized machine consumes more memory than the RAM-optimized machine (i.e., the RAM-optimized machine utilizes less memory).

are remarkably close at 2.04 and 2.016 USD, respectively. The effects of using these machines for the proving phase are depicted in Fig. 10.

Both `snarkjs` and `rapidsnark` demonstrate improved performance on RAM-optimized machines across all circuits. Additionally, they use less RAM on these machines. For instance, when benchmarking the `Exponentiate` circuit on a CPU-optimized machine with an input size of 2^{15}, `rapidsnark` consumes 157 MB of RAM and takes 0.15 seconds. This RAM usage is about 1.4 times higher, and the processing time is approximately 1.2 times slower, compared to the RAM-optimized machine, where `rapidsnark` requires 226 MB and 0.18 seconds for the same input size. In contrast, most other tools display consistent memory consumption regardless of machine type. However, `gnark` deviates from this pattern. For smaller inputs in the `Exponentiate` circuit, `gnark` uses 1.2 times more memory on the CPU-optimized machine (68 MB for 2^{12}) than on its RAM-optimized counterpart (57 MB for 2^{12}). Nevertheless, due to the relatively low RAM requirements for such inputs, this difference might not be significant in real-world applications. A more pronounced difference is evident in `gnark`'s performance on CPU-optimized machines. Its superior performance might indicate efficient parallelization, most notably seen in the `SHA256` circuit (e.g., 2.10 seconds on a CPU-optimized machine for 2^8 vs. 2.99 seconds on a RAM-optimized machine using the BN254 curve). Similarly, both `halo2` and `bellman` exhibit enhanced performance, but their improvements are more modest compared to `gnark`.

6 RUNTIME ESTIMATION

In this section, we harness the data obtained from zk-Bench to accomplish two objectives: *(i)* interpolate and extrapolate the runtimes of arithmetic operations and *(ii)* estimate the runtime of complex proof systems. Our ultimate goal is to offer a user interface that answers user queries about the runtime of cryptographic protocols regardless of their complexity. Furthermore, we want to be able to offer accurate estimation results even if we lack benchmarks for the requested size, or when the cryptographic protocol is in research stage and not yet implemented. We start by outlining our strategy for interpolating and extrapolating the runtime of fundamental arithmetic operation, and then use our techniques to estimate the runtime of entire cryptographic protocols, exemplified by estimating the prover runtime of the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof system.

Arithmetic Data Interpolation and Extrapolation. Given the sparsely sampled benchmark data for a cryptographic operation ϕ, our objective is to derive a function that describes the runtime of ϕ, denoted as $f(x)$. $f(x)$ is designed to estimate the runtime of ϕ for an arbitrary input size n.

For non-amortized operations like scalar multiplication and field addition, we treat them as sequential computations. That is, if a single scalar multiplication consumes x seconds, n such operations will consume $n \cdot x$ seconds. The result can be expressed as: $f(x) = n \cdot x$. However, some complex operations can be amortized, and their performance is non-linear. In such cases:

- If the user queries for a size that lies between two benchmarked intervals, we perform linear interpolation of the two estimates and return the value at the given size.
- If the requested size is outside the tested intervals, we apply linear regression using the least squares method on the nearest four samples, and return the value of the resulting line at the given size.

In cases involving non-linear algorithms, insights into their asymptotic performance can enhance our estimation accuracy. For example, Pippenger's multi-scalar multiplication algorithm is widely used, and its complexity is known to be close to $O(n/\log n)$. Here, to provide better estimates, we use least squares to fit the last 4 samples in the dataset to a function $h(n) = \frac{an+b}{\log n}$ and solve for a and b. Analogous methods are applied for the Cooley-Tukey algorithm [24], which has a complexity of $O(n \log n)$.

To gauge the precision of our estimator, we sampled user queries both within and beyond our benchmark ranges at random samples: for interpolation, at points $2^i + r$ where $i = 10, \ldots, 20$ and r sampled at random in the integer interval $[0, 2^i]$; for extrapolation setting $i = 24$ and taking a random sample as above. The mode of the percent error touches 0.75% for multi-scalar multiplication in BN254, and lower than 0.02% for multi-pairing in bls12-381; the highest recorded error has been of 17.37% for curve25519 on m6i.8xlarge.

Circuit extrapolation. Using the above techniques for estimating the runtime of low-level arithmetic operations, we can move towards *estimating* the running times of entire zero-knowledge proof systems. Given the size of the circuit being proven (in the case of R1CS for instance, the number of addition gates, multiplication gates, instance, and witness sizes) we can provide estimates for the running time of a proof system by enumerating the arithmetic operations being performed, estimating the run-time for each of them, and summing them up.

For the $\mathcal{P}\mathfrak{lon}\mathcal{K}$ proof system, we developed an estimator for the prover's runtime, concentrating primarily on the most computationally intensive operations: multi-scalar multiplications and FFTs. Our observations indicate that even this rudimentary estimation approach can yield sufficiently accurate results for practical user queries. Specifically, the estimated runtimes were consistently within 32% of the actual runtimes of `gnark`'s $\mathcal{P}\mathfrak{lon}\mathcal{K}$ implementation, as illustrated in Table 3.

Table 3. Percent error for estimating the running time of the proving algorithm for the exponentiation circuit prover using $\mathcal{PLONK}$ as implemented in gnark.

Curve	Error	Actual Runtime	Estimated Runtime	Number of gates
BN254	22.40%	3 ms	2.33 ms	10
BN254	6.03%	6 ms	5.64 ms	100
BN254	6.83%	24 ms	25.64 ms	1000
BN254	29.50%	253 ms	178.37 ms	10000
BN254	32.00%	1793 ms	1219.25 ms	100000
BN254	27.66%	12 939 ms	9359.55 ms	1000000
BLS12-381	9.42%	4 ms	3.62 ms	10
BLS12-381	0.20%	9 ms	8.98 ms	100
BLS12-381	16.26%	38 ms	44.18 ms	1000
BLS12-381	22.05%	364 ms	283.73 ms	10000
BLS12-381	21.40%	2445 ms	1921.67 ms	100000
BLS12-381	12.35%	17 141 ms	15 024.86 ms	1000000

We anticipate that analogous techniques can be employed to project the runtime of other proof systems. When complemented with performance data specific to certain implementations, this approach holds the potential to address user queries for proof system circuits of arbitrary size.

7 DISCUSSION

7.1 Limitations

While our study provides valuable insights into the performance of different pulic-key-cryptography-based libraries and tools for ZKP development, several limitations should be acknowledged:

Hardware Considerations. Our evaluation focused on commodity machines, excluding platforms like GPUs, FPGAs, and mobile devices. To fully grasp the real-world applicability of tools for ZKP development, future work should include benchmarks on these platforms.

Naïve public-key operations. As outlined in Sections 2 and 5.3, we are not concerned with the actual representation of elements. Points in an elliptic curve can be represented in multiple ways (depending on the curve, affine points – in Weirstrass, Montgomery, or Edwards form – or projective – in standard, Jacobi, or Chudnovsky coordinates). Additionally, different algorithms might have different constraints: no heap allocations (that's the case for curve25519), constant time, etc. Comparing different algorithms on the same ground is outside the scope of our current arithmetic benchmark. They are meant to give a rough estimate of the runtime of a complex cryptographic algorithm in a usable way. A broader study incorporating these optimizations can elucidate the trade-offs involved. We view an in-depth exploration of the impact of optimizations within across ZKP tools as a promising direction for future work.

Scope of Evaluated Circuits. Our ZKP backend currently only evaluates a limited set of test vectors, and the performance can vary considerably based on the specific implementation of a circuit (cf. Section 5). Given the diversity and complexity of potential ZKP circuits, our chosen circuits might not fully encapsulate all performance scenarios. For instance, in the case of SHA-256, different libraries provided different set of optiomizations, and the *witness generation* in different ZKP tools is sometimes included in the *prove* API, introducing additional complexity in assessing performance on equal grounds.

Nevertheless, the extensibility of zk-Bench ensures ease in integrating additional benchmarks for a more comprehensive evaluation in the future.

7.2 Recommendations

Standardized Test Vectors. When it comes to the test vectors to choose for benchmarks, we find that there is little convergence in the community. Implementations are dispersed, and it is difficult to find implementations that target the same functionality in different tools for ZKP development, such

that equal and fair comparisons can be drawn. We advocate for a focused community discussion on functionalities, emphasizing that a consensus on a benchmark set would enable unbiased third-party performance evaluations.

Interoperable Intermediate Representations. Many libraries provide custom implementations of similar intermediate representations. Whereas initial efforts exist in unifying intermediate representations [63], the degree of interoperability between tools is, thus far, relatively minimal. Evaluating ZKP tooling on common intermediate representation language that compilers can target and libraries can implement, could provide further insights into the performance difference in differing ZKP backends along different test vectors.

Improved Documentation. The implementation of cryptographic protocols and functionalities with ZKP tools remains a complex task. Improved documentation of tooling for ZKPs would be equally beneficial to the practitioners' community, as well as from a benchmarking perspective. For example, the *witness generation* in different ZKP tools is sometimes included in the *prove* API, introducing additional complexity in assessing performance on equal grounds. Similarly, improved documentation of optimizations would tremendously benefit the whole community in qualitatively comparing differing implementations, in addition to quantitative benchmarks.

With zk-Bench, the intuitive frontend for end-users, and the systematization introduced in Section 2, we aim to provide an initial step in addressing the above recommendations. We urge the community to further extend these benchmarking efforts in an open-source effort.

8 RELATED WORK

eBATS [52] is the current largest effort in benchmarking public-key cryptographic operations. However, its focus is limited to Diffie-Hellman, (scalar multiplication), KEM, and sign operations (multiplication and addition) instead of modern cryptography and runtime estimations. Baghery *et al.* [3] examine the setup phase of certain zk-SNARKs, and Botrel *et. al*[16] offer comparative insights into public-key operations in ZKP libraries such as `gnark` and `arkworks`. In contrast to these works, we provide a comprehensive framework that dives deep into both arithmetic operations and the zk-SNARK process, bridging both domains. The drive to standardize ZKPs led to a proposal for a benchmarking framework [10]. This initiative laid some foundational concepts, many of which we've assimilated and expanded upon to deliver an exhaustive and versatile benchmarking tool.

Recently, the practitioners' community has exhibited a growing interest in benchmarking public-key cryptography libraries and tools for ZKP development. Housni has benchmarked many elliptic curve implementations to evaluate the performance of MSMs and pairings. [8] Orthogonally, Bloemen has presented a set of benchmarks for implementations of polynomial commitments. [9] In our work, we build upon these efforts by providing an automated framework to benchmark and compare library implementations across various operations systematically.

Celer Network published a blog post [10] focusing on benchmarking the time and memory costs of proving SHA-256 circuits, which were developed by community members, in various tools for ZKPs. By contrast, our work provides an extensible framework that supports arbitrary circuits, libraries and tools, and thoroughly benchmarks all phases involved in ZKPs. However, the results of this benchmark could heavily depend on how optimized the circuit is. Therefore, we also benchmarked a straightforward exponentiation circuit across the evaluated tools. Moreover, Anoma has developed a framework [11] that benchmarks different ZKP compilation strategies, from libraries used for writing circuits to zkVMs that can run arbitrary programs, with a particular focus on execution

[8] https://hackmd.io/@gnark/eccbench
[9] https://2π.com/23/pc-bench/
[10] https://blog.celer.network/2023/07/14/the-pantheon-of-zero-knowledge-proof-development-frameworks/
[11] https://github.com/anoma/zkp-compiler-shootout

time. Lastly, orthogonal to our work, Delendum has developed a framework for benchmarking zkVMs. [12]

9 CONCLUSION

While the performance measurement of ZKP implementations has long been a relevant topic in academia, its significance has surged recently due to the increased feasibility of proof systems. Further, libraries and tools for ZKP circuits have evolved from proof-of-concepts to robust solutions now used in production. This advancement allows developers to employ innovative tools for expressing computations in a way susceptible to proof systems. However, assessing the performance of a specific implementation, let alone a novel cryptographic protocol, remains a difficult endeavor.

In this paper, we introduced zk-Bench, the first holistic benchmarking framework and estimator tool designed for performance evaluation of public-key cryptography, with a specific focus on practical assessment of general-purpose ZKP systems. zk-Bench is easy to configure and run, whilst being modular enough to include a high variety of real-world applications. Using zk-Bench, we evaluated 13 elliptic curves across 9 libraries and assessed 5 prominent tools for ZKP development with two test vectors — Exponentiation and SHA-256 — uncovering intriguing differences. For example, we find that certain ZKP frameworks favor compute-optimized hardware, while others benefit from memory-optimized hardware. This results in a performance increase of up to 50% for CPU and 40% for memory when computing a proof, guiding developers to select an appropriate tool to develop their ZKP based application.

ACKNOWLEDGMENTS

This work is partially supported by the Center for Responsible, Decentralized Intelligence at Berkeley (Berkeley RDI) and the Ethereum Foundation. The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002. We acknowledge contributions to earlier versions of our open-source repository, zk-Harness, by Celer Network, Morgan Thomas, and bingcicle.

REFERENCES

[1] Mohammad Bagher Abiat. 2023. *Tinybench.* https://github.com/tinylibs/tinybench
[2] arkworks contributors. 2022. `arkworks` *zkSNARK ecosystem.* https://arkworks.rs
[3] Karim Baghery, Axel Mertens, and Mahdi Sedaghat. 2023. Benchmarking the Setup of Updatable zk-SNARKs. *Cryptology ePrint Archive* (2023).
[4] Razvan Barbulescu and Sylvain Duquesne. 2019. Updating key size estimations for pairings. *Journal of cryptology* 32 (2019), 1298–1336.
[5] Paulo SLM Barreto, Ben Lynn, and Michael Scott. 2003. Constructing elliptic curves with prescribed embedding degrees. In *Security in Communication Networks: Third International Conference, SCN 2002 Amalfi, Italy, September 11–13, 2002 Revised Papers 3*. Springer, 257–267.
[6] Paulo SLM Barreto and Michael Naehrig. 2005. Pairing-friendly elliptic curves of prime order. In *International workshop on selected areas in cryptography*. Springer, 319–331.
[7] Eli Ben-Sasson. 2018. *libSTARK.* https://github.com/elibensasson/libSTARK
[8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 701–732.
[9] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013, Part II (LNCS, Vol. 8043)*, Ran Canetti and Juan A. Garay (Eds.). Springer, Heidelberg, 90–108. https://doi.org/10.1007/978-3-642-40084-1_6
[10] Daniel Benarroch, Aurélien Nicolas, Justin Thaler, and Eran Tromer. 2020. ʿCommunity proposal: A benchmarking framework for (zero-knowledge) proof systems. *QEDIT, Tel Aviv-Yafo, Israel, Tech. Rep* (2020).

[12] https://github.com/delendum-xyz/zk-benchmarking

[11] Daniel J Bernstein. 2002. Pippenger's exponentiation algorithm. *Preprint. Available from http://cr. yp. to/papers. html* (2002).

[12] bheisler. 2023. *Rust Criterion.* https://bheisler.github.io/criterion.rs/book/index.html

[13] Nir Bitansky, Ran Canetti, Alessandro Chiesa, Shafi Goldwasser, Huijia Lin, Aviad Rubinstein, and Eran Tromer. 2017. The hunting of the SNARK. *Journal of Cryptology* 30, 4 (2017), 989–1066.

[14] Remco Bloemen. 2023. *Polynomial Commitment Benchmark.*

[15] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. 2022. Gemini: Elastic SNARKs for Diverse Environments. In *EUROCRYPT 2022, Part II (LNCS, Vol. 13276)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Heidelberg, 427–457. https://doi.org/10.1007/978-3-031-07085-3_15

[16] Gautam Botrel and Youssef El Housni. 2023. Faster Montgomery multiplication and Multi-Scalar-Multiplication for SNARKs. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2023), 504–521.

[17] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. 2023. *ConsenSys/gnark: v0.8.0.* https://doi.org/10.5281/zenodo.5819104

[18] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 947–964.

[19] Matteo Campanelli, Nicolas Gailly, Rosario Gennaro, Philipp Jovanovic, Mara Mihali, and Justin Thaler. 2023. Testudo: Linear Time Prover SNARKs with Constant Size Proofs and Square Root Size Universal Setup. In *Progress in Cryptology–Latincrypt 2023: 9th International Conference on Cryptology and Information Security in Latin America.*

[20] David Chaum. 1982. Blind Signatures for Untraceable Payments. In *CRYPTO'82*, David Chaum, Ronald L. Rivest, and Alan T. Sherman (Eds.). Plenum Press, New York, USA, 199–203.

[21] David L Chaum. 1981. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM* 24, 2 (1981), 84–90.

[22] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer, 738–768.

[23] Lambda Class. 2023. *lambdaworks.* https://github.com/lambdaclass/lambdaworks

[24] James W Cooley and John W Tukey. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation* 19, 90 (1965), 297–301.

[25] Whitfield Diffie and Martin E Hellman. 2022. New directions in cryptography. In *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 365–390.

[26] Youssef El Housni. 2021. *Benchmarking pairing-friendly elliptic curves libraries.* https://hackmd.io/@gnark/eccbench

[27] Youssef El Housni and Gautam Botrel. 2022. EdMSM: Multi-Scalar-Multiplication for SNARKs and Faster Montgomery multiplication. *Cryptology ePrint Archive* (2022).

[28] Privacy & Scaling Explorations. 2023. *halo2 Community Edition.* https://github.com/privacy-scaling-explorations/halo2

[29] Facebook. 2023. *winterfell.* https://github.com/facebook/winterfell

[30] Filecoin. 2023. *bellperson.* https://github.com/filecoin-project/bellperson

[31] Georg Fuchsbauer, Michele Orrù, and Yannick Seurin. 2019. Aggregate Cash Systems: A Cryptographic Investigation of Mimblewimble. In *EUROCRYPT 2019, Part I (LNCS, Vol. 11476)*, Yuval Ishai and Vincent Rijmen (Eds.). Springer, Heidelberg, 657–689. https://doi.org/10.1007/978-3-030-17653-2_22

[32] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive* (2019).

[33] Steven D Galbraith, Kenneth G Paterson, and Nigel P Smart. 2008. Pairings for cryptographers. *Discrete Applied Mathematics* 156, 16 (2008), 3113–3121.

[34] Rosario Gennaro and Daniel Wichs. 2013. Fully Homomorphic Message Authenticators. In *ASIACRYPT 2013, Part II (LNCS, Vol. 8270)*, Kazue Sako and Palash Sarkar (Eds.). Springer, Heidelberg, 301–320. https://doi.org/10.1007/978-3-642-42045-0_16

[35] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC '85)*. Association for Computing Machinery, 291–304.

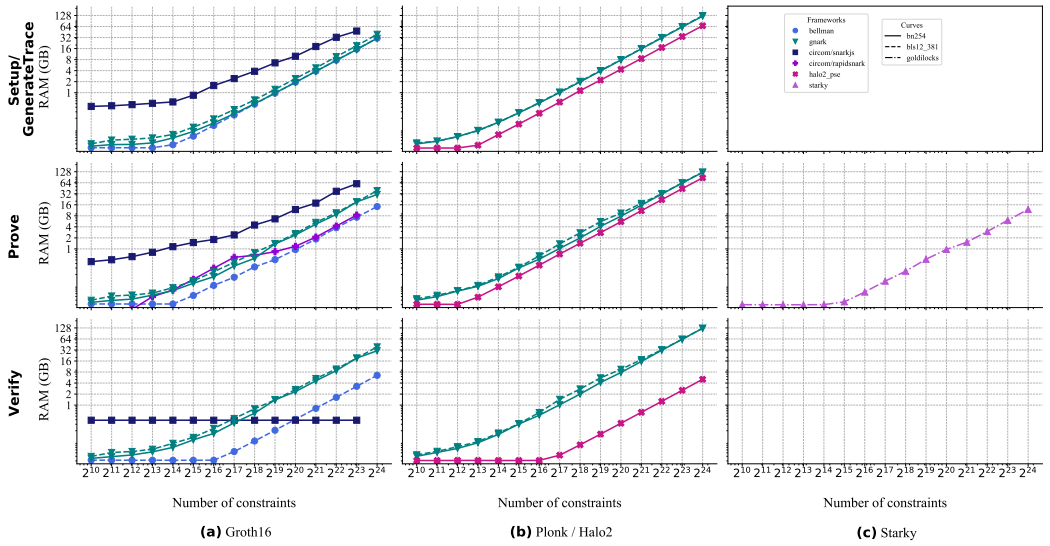
[36] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 305–326.

[37] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2016. Zcash protocol specification. *GitHub: San Francisco, CA, USA* 4 (2016), 220.

[38] iden3. 2021. *snarkjs.* https://github.com/iden3/snarkjs

[39] iden3. 2023. *Circom: Circuit Compiler For ZK Proving Systems.* https://github.com/iden3/circom

[40] Taechan Kim and Razvan Barbulescu. 2016. Extended tower number field sieve: A new complexity for the medium prime case. In *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*. Springer, 543–571.
[41] Ahmed Kosba. 2022. *jsnark*. https://github.com/akosba/jsnark
[42] Matter Labs. 2023. *bellman Community Edition*. https://github.com/matter-labs/bellman
[43] Matter Labs. 2023. *Boojum*. https://github.com/matter-labs/era-boojum
[44] O1 labs. 2023. *kimchi*. https://github.com/o1-labs/proof-systems
[45] O1 labs. 2023. *snarky*. https://github.com/o1-labs/snarky
[46] O1 labs. 2023. *snarkyjs*. https://github.com/o1-labs/snarkyjs
[47] Patrick Longa. 2023. Efficient Algorithms for Large Prime Characteristic Fields and Their Application to Bilinear Pairings. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2023), 445–472.
[48] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2111–2128.
[49] Alfred Menezes, Palash Sarkar, and Shashank Singh. 2017. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Paradigms in Cryptology–Mycrypt 2016. Malicious and Exploratory Cryptology: Second International Conference, Mycrypt 2016, Kuala Lumpur, Malaysia, December 1-2, 2016, Revised Selected Papers*. Springer, 83–108.
[50] Alfred Menezes, Scott Vanstone, and Tatsuaki Okamoto. 1991. Reducing Elliptic Curve Logarithms to Logarithms in a Finite Field. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing* (New Orleans, Louisiana, USA) *(STOC '91)*. Association for Computing Machinery, New York, NY, USA, 80–89. https://doi.org/10.1145/103418.103434
[51] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. 2022. Powers-of-Tau to the People: Decentralizing Setup Ceremonies. *Cryptology ePrint Archive* (2022).
[52] European Network of Excellence for Cryptology. 2006. *eBATS: ECRYPT Benchmarking of Asymmetric Systems*. https://www.ecrypt.eu.org/ebats/
[53] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*. 2075–2092.
[54] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 238–252. https://doi.org/10.1109/SP.2013.47
[55] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2016. Pinocchio: Nearly practical verifiable computation. *Commun. ACM* 59, 2 (2016), 103–112.
[56] Geovandro CCF Pereira, Marcos A Simplício Jr, Michael Naehrig, and Paulo SLM Barreto. 2011. A family of implementation-friendly BN elliptic curves. *Journal of Systems and Software* 84, 8 (2011), 1319–1326.
[57] Nicholas Pippenger. 2022. A Formula for the Determinant. *CoRR* abs/2206.00134 (2022). https://doi.org/10.48550/arXiv.2206.00134 arXiv:2206.00134
[58] Plonky3. 2023. *Plonky3*. https://github.com/Plonky3/Plonky3
[59] Mir Protocol. 2023. *Plonky2*. https://github.com/mir-protocol/plonky2
[60] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*. IEEE, 459–474.
[61] scipt lab. 2020. *libsnark*. https://github.com/scipr-lab/libsnark
[62] scipt lab. 2021. *libiop*. https://github.com/scipr-lab/libiop
[63] Srinath Setty, Justin Thaler, and Riad Wahby. 2023. Customizable constraint systems for succinct arguments. *Cryptology ePrint Archive* (2023).
[64] Espresso Systems. 2023. *jellyfish*. https://github.com/EspressoSystems/jellyfish
[65] Psi Vesely, Kobi Gurkan, Michael Straka, Ariel Gabizon, Philipp Jovanovic, Georgios Konstantopoulos, Asa Oines, Marek Olszewski, and Eran Tromer. 2022. Plumo: An ultralight blockchain client. In *International Conference on Financial Cryptography and Data Security*. Springer, 597–614.
[66] Lawrence C Washington. 2008. *Elliptic curves: number theory and cryptography*. CRC press.
[67] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 3003–3017.
[68] ZCash. 2023. *halo2*. https://github.com/zcash/halo2
[69] zkcrypto. 2023. *bellman: zk-SNARK library*. https://github.com/zkcrypto/bellman



(a) Groth16 **(b)** Plonk / Halo2 **(c)** Starky

Fig. 11. Memory consumption of Exponentiate circuit on m6i.8xlarge. The memory consumption of Circom is constant for verification as the API provided by Circom demands for deserializatioin of verification key, public, and proof, which are relatively small. For other ZKP tools and libraries, the API demands for deserialization of parameter files that grow in size with size of the circuit. Memory consumption for serialization of starky is missing due to non-existent serialization APIs, at the time of writing.

A SNARK AND STARK LIBRARIES

Table 4. Overview of the popularity of common SNARK and STARK frameworks. **(F)**rontend and **(B)**ackend of related frameworks are grouped together. Updated as of 1st of August 2023.

Project	F	B	# Stars	Last Update
libsnark [61]			1668	07/2020
jsnark [41]			191	12/2022
snarkjs [38]			1471	06/2023
circom [39]			921	06/2023
gnark [17]			1034	07/2023
Bellman / zkcrypto [69]			814	03/2023
arkworks [2]			713	05/2023
Winterfell / Meta [29]			623	07/2023
plonky2 [59]			534	07/2023
snarky [45]			496	05/2023
snarkyjs [46]			338	07/2023

Project	F	B	# Stars	Last Update
Kimchi / O(1) Labs [44]			299	07/2023
libstark [7]			448	11/2018
halo2 / ZCash [68]			457	07/2023
lambdaworks [23]			350	07/2023
jellyfish / Espresso Systems [64]			271	07/2023
boojum [43]			170	07/2023
bellperson / filecoin [30]			158	07/2023
bellman_ce / matter labs [42]			138	04/2023
libiop [62]			127	05/2021
plonky3 [58]			138	07/2023
halo2_ce [28] / PSE			116	07/2023

Table 4 presents a comprehensive overview of the most prevalent SNARK and STARK frameworks used in the development of Zero-Knowledge Proof (ZKP) programs. Each toolchain typically encompasses both a front-end and a back-end, often provided by the same framework. However, there are exceptions to this pattern, such as iden3's Circom project. In this case, Circom serves as the front-end, while either snarkjs or rapidsnark (offering only proving capabilities) function as the back-ends.

Beyond showcasing the structural composition of these toolchains, Table 4 offers insights into each project's popularity and activity level. This is gauged through the number of GitHub stars

Table 5. Coefficient of Variation (CV) for multiplication and addition in BN254 and BLS12-381. Distributions of CV Ÿ 1 can be considered low-variance, while those with CV ¡ 1 can be considered high variance.

machine	library	curve	operation	cv
m5.2xlarge	arkworks_curves	bls12_381	add	0.117896
	zkcrypto	bls12_381	mul	0.149145
			add	0.145491
	pairing_ce	bn254	mul	0.186713
			add	0.448729
		bls12_381	add	0.420041
	halo2_curves	bn254	mul	0.173454
	pairing_ce	bls12_381	mul	0.085221
	blstrs	bls12_381	mul	0.614840
			add	0.604633
	arkworks_curves	bn254	mul	0.071497
			add	0.085387
		bls12_381	mul	0.521088
	halo2_curves	bn254	add	0.227429
m5.large	pairing_ce	bn254	add	2.528118
	zkcrypto	bls12_381	mul	2.482617
			add	2.054006
	pairing_ce	bn254	mul	4.200262
		bls12_381	mul	4.407481
	blstrs	bls12_381	mul	4.097602
	halo2_curves	bn254	mul	4.059257
			add	1.510318
	blstrs	bls12_381	add	0.778731
	pairing_ce	bls12_381	add	1.366626
m6g.8xlarge	pairing_ce	bls12_381	add	0.351906
	zkcrypto	bls12_381	mul	0.161643
			add	0.106811
	pairing_ce	bn254	mul	0.032653
			add	0.044894
		bls12_381	mul	0.030930
	halo2_curves	bn254	mul	0.021249
	arkworks_curves	bls12_381	mul	0.045578
	blstrs	bls12_381	mul	1.417817
			add	0.066837
	arkworks_curves	bn254	mul	0.050363
			add	0.082494
		bls12_381	add	0.426573
	halo2_curves	bn254	add	0.104693
m6i.8xlarge	pairing_ce	bn254	mul	0.217116
			add	0.198308
		bls12_381	mul	0.210383
			add	0.206211
	halo2_curves	bn254	mul	0.390493
			add	0.074193
	arkworks_curves	bn254	mul	0.421921
	blstrs	bls12_381	add	0.061124
	arkworks_curves	bn254	add	0.065893
		bls12_381	mul	0.071483
			add	0.063589
	zkcrypto	bls12_381	add	0.075747
	blstrs	bls12_381	mul	0.146054
	zkcrypto	bls12_381	mul	0.300691

each project has accumulated - a measure of community endorsement - as well as the date of the last commit, indicating the project's recent activity and ongoing maintenance efforts.

B DATA VALIDATION

To validate the data we obtained from the respective machines, we measure the standard deviation μ and mean σ of all operations to calculate the *coefficient of variation*, a statistical measure that is defined as the ratio of the standard deviation to the mean, allowing us to compare the relative variability across datasets and machines. We present the coefficient of variation for field addition and multiplication in Table 5, and the coefficient of variation for MSM on AWS m6i.8xlarge in Table 6. We find that the mean of the coefficient of variation is ≈ 0.39, which indicates relatively low variability We observe, that especially the smallest instance, m5.large, incurs a large coefficient of variation across different libraries and operations, which indicates that the samples taken from this machine are relatively inconsistent. For MSM, we observe the mean of the coefficient of variation to be ≈ 1.34 for MSM in G_1 and ≈ 1.02 for MSM in G_2 respectively, which signifies larger variability in the sampled data. We urge for further investigation to draw conclusions on whether this variability is introduced by on-demand server infrastructure.

Table 6. Coefficient of Variation (CV) for multiplication and addition in BN254 and BLS12-381. Distributions of CV Ÿ 1 can be considered low-variance, while those with CV ¡ 1 can be considered high variance.

machine	library	curve	operation	size	cv
m6i.8xlarge	arkworks	bls12_381	msm_G1	2	1.023519
m6i.8xlarge	arkworks	bls12_381	msm_G1	4	1.032269
m6i.8xlarge	arkworks	bls12_381	msm_G1	8	0.670592
m6i.8xlarge	arkworks	bls12_381	msm_G1	16	0.925825
m6i.8xlarge	arkworks	bls12_381	msm_G1	2	1.319842
m6i.8xlarge	arkworks	bls12_381	msm_G1	64	1.005520
m6i.8xlarge	arkworks	bls12_381	msm_G1	128	1.536985
m6i.8xlarge	arkworks	bls12_381	msm_G1	256	0.929388
m6i.8xlarge	arkworks	bls12_381	msm_G1	512	1.158935
m6i.8xlarge	arkworks	bls12_381	msm_G1	1024	1.261890
m6i.8xlarge	arkworks	bls12_381	msm_G1	2048	1.400382
m6i.8xlarge	arkworks	bls12_381	msm_G1	4096	1.264280
m6i.8xlarge	arkworks	bls12_381	msm_G1	8192	2.557883
m6i.8xlarge	arkworks	bls12_381	msm_G1	16384	3.694018
m6i.8xlarge	arkworks	bls12_381	msm_G1	2768	2.656017
m6i.8xlarge	arkworks	bls12_381	msm_G1	65536	0.207187
m6i.8xlarge	arkworks	bls12_381	msm_G1	131072	0.402804
m6i.8xlarge	arkworks	bls12_381	msm_G1	262144	1.885476
m6i.8xlarge	arkworks	bls12_381	msm_G1	524288	1.218963
m6i.8xlarge	arkworks	bls12_381	msm_G1	1048576	1.294875
m6i.8xlarge	arkworks	bls12_381	msm_G1	2097152	1.346189
m6i.8xlarge	blstrs	bls12_381	msm_G1	2	0.044488
m6i.8xlarge	blstrs	bls12_381	msm_G1	4	0.390781
m6i.8xlarge	blstrs	bls12_381	msm_G1	8	0.056672
m6i.8xlarge	blstrs	bls12_381	msm_G1	16	0.043823
m6i.8xlarge	blstrs	bls12_381	msm_G1	2	0.359884
m6i.8xlarge	blstrs	bls12_381	msm_G1	64	0.303458
m6i.8xlarge	blstrs	bls12_381	msm_G1	128	0.472966
m6i.8xlarge	blstrs	bls12_381	msm_G1	256	0.922661
m6i.8xlarge	blstrs	bls12_381	msm_G1	512	0.609237
m6i.8xlarge	blstrs	bls12_381	msm_G1	1024	2.029009
m6i.8xlarge	blstrs	bls12_381	msm_G1	2048	0.741856
m6i.8xlarge	blstrs	bls12_381	msm_G1	4096	0.366328
m6i.8xlarge	blstrs	bls12_381	msm_G1	8192	0.289558
m6i.8xlarge	blstrs	bls12_381	msm_G1	16384	0.687239
m6i.8xlarge	blstrs	bls12_381	msm_G1	2768	0.081350
m6i.8xlarge	blstrs	bls12_381	msm_G1	65536	0.130423
m6i.8xlarge	blstrs	bls12_381	msm_G1	131072	0.138213
m6i.8xlarge	blstrs	bls12_381	msm_G1	262144	0.533823
m6i.8xlarge	blstrs	bls12_381	msm_G1	524288	3.483468
m6i.8xlarge	blstrs	bls12_381	msm_G1	1048576	4.161291
m6i.8xlarge	blstrs	bls12_381	msm_G1	2097152	2.618671
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	1	0.865825
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	2	0.982300
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	4	0.835601
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	8	0.717361
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	16	1.127157
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	2	1.942223
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	64	7.285463
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	128	0.151263
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	256	3.089730
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	512	2.877493
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	1024	0.285611
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	2048	0.231980
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	4096	0.353366
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	8192	0.099853
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	16384	0.171358
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	2768	0.058058
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	65536	0.147453
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	131072	0.034668
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	262144	0.081413
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	524288	0.366701
m6i.8xlarge	pairing_ce	bls12_381	msm_G1	1048576	0.724438
m6i.8xlarge	arkworks	bn254	msm_G1	2	0.848204

machine	library	curve	operation	size	cv
m6i.8xlarge	arkworks	bn254	msm_G1	4	1.886335
m6i.8xlarge	arkworks	bn254	msm_G1	8	1.301629
m6i.8xlarge	arkworks	bn254	msm_G1	16	0.869819
m6i.8xlarge	arkworks	bn254	msm_G1	2	1.657673
m6i.8xlarge	arkworks	bn254	msm_G1	64	1.091215
m6i.8xlarge	arkworks	bn254	msm_G1	128	0.864722
m6i.8xlarge	arkworks	bn254	msm_G1	256	0.582581
m6i.8xlarge	arkworks	bn254	msm_G1	512	3.099921
m6i.8xlarge	arkworks	bn254	msm_G1	1024	3.295864
m6i.8xlarge	arkworks	bn254	msm_G1	2048	0.941437
m6i.8xlarge	arkworks	bn254	msm_G1	4096	0.878906
m6i.8xlarge	arkworks	bn254	msm_G1	8192	1.770486
m6i.8xlarge	arkworks	bn254	msm_G1	16384	0.505038
m6i.8xlarge	arkworks	bn254	msm_G1	2768	0.247747
m6i.8xlarge	arkworks	bn254	msm_G1	65536	0.163440
m6i.8xlarge	arkworks	bn254	msm_G1	131072	2.003160
m6i.8xlarge	arkworks	bn254	msm_G1	262144	1.005785
m6i.8xlarge	arkworks	bn254	msm_G1	524288	2.797046
m6i.8xlarge	arkworks	bn254	msm_G1	1048576	0.969073
m6i.8xlarge	arkworks	bn254	msm_G1	2097152	4.846506
m6i.8xlarge	arkworks	bn254	msm_G1	2	0.842366
m6i.8xlarge	arkworks	bn254	msm_G1	4	1.521145
m6i.8xlarge	arkworks	bn254	msm_G1	8	1.372733
m6i.8xlarge	arkworks	bn254	msm_G1	16	1.068469
m6i.8xlarge	arkworks	bn254	msm_G1	2	1.414223
m6i.8xlarge	arkworks	bn254	msm_G1	64	1.190394
m6i.8xlarge	arkworks	bn254	msm_G1	128	1.026420
m6i.8xlarge	arkworks	bn254	msm_G1	256	0.478278
m6i.8xlarge	arkworks	bn254	msm_G1	512	1.378518
m6i.8xlarge	arkworks	bn254	msm_G1	1024	2.168509
m6i.8xlarge	arkworks	bn254	msm_G1	2048	1.135775
m6i.8xlarge	arkworks	bn254	msm_G1	4096	0.777476
m6i.8xlarge	arkworks	bn254	msm_G1	8192	0.947085
m6i.8xlarge	arkworks	bn254	msm_G1	16384	0.219800
m6i.8xlarge	arkworks	bn254	msm_G1	2768	0.171549
m6i.8xlarge	arkworks	bn254	msm_G1	65536	0.579498
m6i.8xlarge	arkworks	bn254	msm_G1	131072	0.471684
m6i.8xlarge	arkworks	bn254	msm_G1	262144	1.522305
m6i.8xlarge	arkworks	bn254	msm_G1	524288	3.603968
m6i.8xlarge	arkworks	bn254	msm_G1	1048576	1.214127
m6i.8xlarge	arkworks	bn254	msm_G1	2097152	4.754687
m6i.8xlarge	pairing_ce	bn254	msm_G1	1	0.641293
m6i.8xlarge	pairing_ce	bn254	msm_G1	2	0.602851
m6i.8xlarge	pairing_ce	bn254	msm_G1	4	0.911040
m6i.8xlarge	pairing_ce	bn254	msm_G1	8	0.890320
m6i.8xlarge	pairing_ce	bn254	msm_G1	16	1.223913
m6i.8xlarge	pairing_ce	bn254	msm_G1	2	1.186686
m6i.8xlarge	pairing_ce	bn254	msm_G1	64	1.501231
m6i.8xlarge	pairing_ce	bn254	msm_G1	128	4.109449
m6i.8xlarge	pairing_ce	bn254	msm_G1	256	5.544364
m6i.8xlarge	pairing_ce	bn254	msm_G1	512	3.752064
m6i.8xlarge	pairing_ce	bn254	msm_G1	1024	2.953111
m6i.8xlarge	pairing_ce	bn254	msm_G1	2048	0.085942
m6i.8xlarge	pairing_ce	bn254	msm_G1	4096	3.174368
m6i.8xlarge	pairing_ce	bn254	msm_G1	8192	0.116408
m6i.8xlarge	pairing_ce	bn254	msm_G1	16384	0.055445
m6i.8xlarge	pairing_ce	bn254	msm_G1	2768	0.091232
m6i.8xlarge	pairing_ce	bn254	msm_G1	65536	0.070121
m6i.8xlarge	pairing_ce	bn254	msm_G1	131072	0.061525
m6i.8xlarge	pairing_ce	bn254	msm_G1	262144	0.087837
m6i.8xlarge	pairing_ce	bn254	msm_G1	524288	0.243712
m6i.8xlarge	pairing_ce	bn254	msm_G1	1048576	0.138218