Ligetron: Lightweight Scalable End-to-End Zero-Knowledge Proofs Post-Quantum ZK-SNARKs on a Browser

Ruihan Wang
Ligero Inc.
ruihan@ligero-inc.com

Carmit Hazay
Ligero Inc.
carmit@ligero-inc.com

Muthuramakrishnan Venkitasubramaniam
Ligero Inc.
muthu@ligero-inc.com

***Abstract*—Zero-Knowledge (ZK) proofs were introduced in the seminal work of Goldwasser, Micali, and Rackoff (STOC 1985) and remain one of the cornerstones of modern cryptography. With the advent of Blockchains, there has been reinvigorated interest in deploying ZK-proof systems in the form of ZK-SNARKs. ZKSNARKs are an attractive variant as they are non-interactive (in fact, publicly verifiable) and succinct. Yet, current deployments require huge running times and/or very large memory, and scaling them to large circuits cannot be accomplished on commodity hardware. We design and implement an efficient sublinear non-interactive zero-knowledge system, Ligetron that can be deployed as a web application and scales to billions of gates. Core to our construction is identifying a good intermediate representation, namely Web Assembly (WASM) that is: (1) versatile to represent complex computations, (2) can be compiled from most popular high-level languages, and (3) embodies rich semantics to derive space-efficiency. On the backend, we design and implement a space-efficient variant of the Ligero ZK system introduced in the work of Ames et al. (ACM CCS 2017) that can leverage the semantics of WASM. Ligetron is the first post-quantum ZK-SNARK to scale to billion gates and run from a browser. On commodity hardware, Ligetron scales to arbitrarily large circuits while showcasing competitive prover/verifier running times and better proof lengths than all previous post-quantum ZK-SNARKs.**

***Index Terms*—ZK-SNARKs, space-efficient, browser-based, post-quantum, zero-knowledge**

1. Introduction

Zero-knowledge proofs, introduced by Goldwasser, Micali, and Rackoff [GMR85] are powerful cryptographic objects that allow a prover to convince a verifier of a statement while revealing nothing beyond the validity of the statement. Succinct non-interactive zero-knowledge arguments (ZK-SNARKs and ZK-SNARGs) are variants of zero-knowledge proof systems that offer very efficient verification, namely, proof lengths and verification times that are polylogarithmic in the size of the instance. ZK-SNARKs have been the focus of intense research from both theory and practice in the past few years as they are becoming an indispensable tool for bringing privacy and efficiency to blockchains (see [Ish20], [Tha21] for two recent surveys).

While the initial constructions of concretely efficient ZK-SNARKs suffered from significantly high prover times, recent works have shown how to improve the computational complexity to essentially linear in the time taken to compute the underlying relation (for an NP-language) [BBB+18], [XZZ+19], [ZXZS20], [Set20], [SL20], [KMP20], [BCL20], [LSTW21]. However, these works come with a steep price in terms of *space*, namely, for computations that take time T, the space complexity of the prover is $\Omega(T)$. In theory, it is possible to achieve asymptotic efficiency in both time and space [BGH19], [COS20], [BCMS20], [BHR+20], [BHR+21], [BBHV22], however, these works require heavy use of public-key operations and/or complex compilation of the original statement and consequently, these systems have either not been deployed in practice or incur big overheads. This fact turns out to be a major bottleneck in scaling up zero-knowledge proofs to large computations. In fact, there are currently *no* implementations of non-interactive zero-knowledge proofs (let alone sublinear) that can prove a statement with billion constraints on commodity hardware. In the interactive setting, recent works [YSWW21], [WYY+22] have been able to scale zero-knowledge to trillion constraints. These works are not succinct, not publicly verifiable (i.e. designated verifier), and require multiple rounds of communication between the prover and verifier.

In this work, we present an *end-to-end* zero-knowledge proof system, that simultaneously achieves the following features:

- *Non-interactive* - Ligetron is publicly verifiable.
- *Sublinear* - in fact, our proof system has shorter proofs than all previous ZK-SNARKS based on symmetric-key primitives up to a billion gates.
- *Plausibly post-quantum and transparent* - Ligetron relies on collision-resistant hash-functions in a *black-box* manner and requires no trusted setup.
- *Space-efficient* - Ligetron requires space proportional to $O(\max(\sqrt{\text{circuit size}}, S))$ where S is the space required to directly evaluate the circuit or instance.
- *Lightweight* - our end-to-end prover running times are competitive against all state-of-the-art ZK-SNARKs.

In other words, Ligetron achieves the best post-quantum ZK-SNARK in prover/verifier time and proof length that can scale to billions of gates. We implemented our system and demonstrate its concrete efficiency. To exemplify the space efficiency of our system, we cross-compiled our prover and verifier code to run from a desktop browser and tested up to a billion (roughly 2^{30}) gates. In comparison, the largest circuit benchmarked by previous works is smaller and require heavy hardware. For example, Brakedown [GLS+21a] scales to 2^{27} gates on 64-core, 128GB machine while Orion [XZS22] scales to a similar size on a 128-core, 512GB RAM. Browsers, on the other hand, have restricted memory allowance - up to 4GB on desktops and 1GB on smartphones.

Performance. We implemented Ligetron on C++ and benchmarked its performance both on AWS `c6i.2xlarge` (16GB RAM and 4 CPU cores) and a browser. We measure the end-to-end performance of our system. More precisely, our prover algorithm is not provided with a flat arithmetic circuit (or R1CS) but instead provided a succinct WASM code representing the NP relation, which is expanded out to a circuit on the fly by the prover and verifier. In particular, our prover and verifier timing includes the time for circuit/constraint generation. Our prover runs 500ns/gate on `c6i` and 3.5μs/gate on a browser. The verifier, on the other hand, takes 250ns/gate on `c6i` and 1.5μs/gate on a browser. In the special case of the batched setting where the prover proves several instances of the same relation, our prover and verifier speeds are 60ns/gate and 3ns/gate on a `c6i` and 1μs/gate and 15ns/gate on the browser, respectively. Ligetron achieves better proof length compared to all concretely efficient post-quantum ZK-SNARKs for medium to large circuits. We ran experiments up to a billion gates on the browser and 10 billion on `c6i`. Indeed, our system can scale to arbitrarily large circuits as the prover and verifier only require a few 100s of MB of memory for structured computations.

1.1. Related Works

We refer the reader to [Ish20], [Tha21] for two recent excellent surveys on the state-of-the-art. We present related works that are most relevant to our system, specifically, only those that are concretely efficient. In fact, the most recent and popular constructions can be distilled to the following paradigm: (1) Build an interactive oracle proof (IOP) whose soundness holds information theoretically, and (2) Compile to a ZK-SNARK using an appropriate cryptographic commitment and make it non-interactive via the Fiat-Shamir heuristic.

Post-quantum schemes. All known concretely efficient post-quantum ZK-SNARKs rely only on collision-resistant hash-functions to implement the proof oracle via Merkle trees [AHIV17], [BBHR19], [ZXZS20], [BFH+20], [GLS+21b], [XZS22]. These constructions can be further classified into three types: (1) Sum-check based [ZXZS20], [GLS+21b], (2) FRI-based [BBHR19], and (3) MPC-in-the-head based [AHIV17]. Some works rely on the composition of multiple protocols to optimize prover complexity and/or proof length [BFH+20], [XZS22]. Our work falls in the third category and is a slight variant of the scheme in [AHIV17]. Among the post-quantum schemes, the Breakdown/Shockwave [GLS+21b] systems and Orion [XZS22] are the most recent that, besides demonstrating proof systems with asymptotic *linear-time* prover complexity, provide implementations with the fastest prover running time and shortest proofs to-date. In particular, both these systems have running times of roughly 1μs per gate when running on hardware with reasonably large RAM.

Pre-quantum schemes. ZK-SNARKs based on public-key assumptions (that are not post-quantum secure) include [MBKM19], [CHM+20], [GWC19] which rely on univariate polynomials and [WTS+18], [XZZ+19], [Set20] that rely on multivariate linear polynomials. All schemes in this category require a setup (some only require a *universal* setup) and typically have prover complexities that are an order of magnitude slower than the post-quantum counterparts primarily because public-key operations (i.e. multiexponentiations) are expensive. However, the systems in this design are most popular owing to the super succinct proof lengths (a few hundred bytes). In comparison, the post-quantum constructions have proof lengths in 10s of kilobytes.

Interactive, i.e., non-publicly-verifiable schemes. A recent line of work has focused on designing highly efficient zero-knowledge proof systems in the interactive setting [WYY+22], [WYKW21], [BMRS21]. These systems showcase the fastest provers/verifiers to date among any zero-knowledge system (SNARKs and otherwise) with speeds around 100ns/gate. Furthermore, these works are also space efficient and have managed to scale their systems to run trillions of gates [WYKW21]. However, these works do not have the desirable public verifiability property and have linear circuit size-proof lengths.

1.2. Our Techniques

Succinctly representing the NP relation. Rank-1-Constraint-Systems[1] (R1CS) has become a *de-facto* standard for representing NP statements for ZK-SNARKs. While this representation is very simple and close to a plain description of an arithmetic circuit, it does not retain any underlying semantic structure of the original application to facilitate space efficiency. Our approach is to avoid ahead-of-time circuit generation by generating the circuit and constraints on the fly at runtime. In our solution, we chose an intermediate representation that retains sufficient high-level structure while being adequately low-level (simple) to connect our prover/verifier algorithms: WebAssembly (WASM). The biggest advantage of WASM compared to R1CS is that it

1. An R1CS instance is a tuple $(\mathbb{F}, A, B, C, M)$, where $A, B, C \in \mathbb{F}^{M\times M}$ and the solution is a vector $w \in \mathbb{F}^M$ such that $A\cdot z \circ B\cdot z = C\cdot z$ where $\circ$ denotes Hadamard product and $\mathbb{F}$ is a finite field.

retains sufficient high-level semantics. Specifically, it retains information about scoping and control structure which is important in space efficiency (i.e. garbage collection). Although WASM was designed for high-performance code execution on browsers, it also can be used as a standalone runtime outside the Web environment. Owing to WASM's clean semantics and a minimal runtime environment, we were able to design a proof system that connects smoothly with the WebAssembly's execution model.

While most zero-knowledge implementations use R1CS as the intermediate representation, there are a few exceptions that use customized virtual machines. The RISCZERO system provides an implementation of a Virtual Machine for ZK that is based on RISC-V architecture [ris]. Compared to WebAssembly's fixed-length instructions and stack-based design, the RISC architecture is based on variable-length instructions and registers. RISC-V architecture is more complex than WASM and it is not possible to track the lifetime of variables. Furthermore, control instructions in RISC are low-level and unstructured which makes it hard to use any backend-specific optimizations. One advantage of WASM with other architectures pursued in ZK implementations is the wide availability of existing toolchains. With the growing adoption of WASM, the effort to compile legacy code becomes simple. In contrast, other systems will either require developing a compiler plugin or designing a special-purpose DSL.

A few ZK implementations have been deployed on the browser (See [Mat], [QED] and references therein) but they have not been tested/deployed to large circuits. Almost all known implementations rely on ZK-SNARKs that are space inefficient and rely on public-key assumptions. We are not aware of any implementation based on symmetric-key primitives.

In contrast, our WASM code is succinct in that it does not depend on the "input" length. For example, our SHA benchmark uses a fixed length WASM code for computing SHA on any length.

Ligetron IOP. Our backend ZKSNARK system is based on the Ligero proof system developed by Ames et al. [AHIV17]. We begin with a high-level description of the Ligero proof system in the IOP model and identify the bottlenecks to simultaneously achieve time and space efficiency.

Given an arithmetic circuit C over a field $\mathbb{F}$, the Ligero system proves satisfiability of C as follows:

1) **Stage 1: Preparing the proof oracle.** In the first step in Ligero, the prover computes an "extended" witness (of size $O(|\mathrm{C}|)$) that incorporates all intermediate computations (namely, the output of each gate) and encodes it using an Interleaved Reed-Solomon code. This code is set as the proof oracle.
2) **Stage 2: Testing codes/computation.**
   a) **Testing the encoding:** Next, the verifier tests if the prover set the oracle with a valid encoding of values. The Interleaved Reed-Solomon Code can be interpreted as a matrix U where each row is a Reed-Solomon code of some message (see Section 3.1). The verifier challenges the prover with a set of random elements (one for each row of U) and the prover responds with a random linear combination of the rows based on the randomness provided by the verifier. The prover can of course lie about the response, but, such behavior will be caught in the column check. The verifier rejects if this combination is not a valid (Reed Solomon) code. The idea is that if each row of U is a valid Reed Solomon code, then by linearity the random linear combination provided by the prover must also be a valid Reed Solomon code.
   b) **Testing linear constraints:** Linear constraints incorporate all the addition gates and circuit wiring in C. The verifier tests these constraints by providing randomness and obtaining an encoding of a random linear combination of the result of all the linear constraints applied to the extended witness. Given the prover's response, the verifier checks if the response encodes values that sum up to 0. The idea here is that even if one of the linear relations does not hold, then the values encoded in the random linear combination will not sum up to 0 with a very high probability.
   c) **Testing quadratic constraints:** Quadratic constraints incorporate all the multiplication gates in C. The verifier tests these constraints analogously to the linear constraints. Specifically, the verifier checks if the prover's response encodes a vector of all zeros. This test utilizes the strong multiplicative property of the Reed-Solomon encoding [RS60].
3) **Stage 3: Column check.** Finally, the verifier checks if the responses provided by the prover in the three tests presented above are consistent with the code in the proof oracle. Since all the tests can be performed via row operations on the matrix, the verifier selects a random subset of the columns of the matrix and recomputes the results of the tests for these columns, and checks if they are consistent with the responses.

Compiling the IOP system to a sublinear argument is achieved by replacing the proof oracle with the root hash of a Merkle hash tree with leaves as the elements of the code matrix and providing Merkle decommitments along with the elements (columns) revealed in the column check step [Kil92], [BCS16].

Next, we analyze the space complexity of the Ligero system.

1) The first step of the argument system involves the prover computing the code generated by encoding the witness where this codeword serves as the proof oracle. This is followed by computing the Merkle hash tree of the code. The size of the code is $O(|\mathrm{C}|)$ for circuit C and if we naively compute the Merkle tree it will require holding the entire code in memory. However, if the Interleaved Reed Solomon code can be generated one row at a time then the Merkle hash tree can be computed with space proportional to the code length (i.e. number of columns in the matrix) as the hash

of the leaves can be iteratively aggregated using the Merkle-Damgard construction [CDMP05]. We remark that computing the code one row at a time is not straightforward as the Ligero proof system actually requires the extended witness to be arranged in a specific structure. The verifier on the other hand can check the Merkle decommitments of the κ columns in space that is linear in the security parameter κ and logarithmic in code length.

2) In the code test, the prover computes a random linear combination of the rows of the matrix. Once again, if we assume that the code matrix can be computed one row at a time, then the linear combination can also be computed in space proportional to the length of the code by maintaining a running aggregate.
3) The linear test is one of the main bottlenecks in terms of space complexity. As the wiring in the circuit can be arbitrary, the linear constraints can involve values encoded in arbitrary rows of the matrix. This means that even if the code can be computed one row at a time, computing the response to the linear constraints could involve recomputing the entire code for each constraint to access different rows of the code and this blows up the running time of the prover beyond quasi-linear in the worst case. The issue of the wiring in the circuit C being arbitrary (as described in the previous step) poses a challenge to improving the verifier's space complexity as well. The main technical challenge at the backend is computing the response for the linear test in a space-efficient manner. We describe below our solution that takes advantage of the WASM semantics.
4) In the quadratic test, the verifier checks the correctness of all the multiplication gates. The prover prepares the extended witness in a specific way where the multiplication gates are batched and the wire values are aligned so that they can be tested for correctness as follows: the verifier provides randomness and the prover provides an aggregate computed via row operations which the verifier checks if it encodes the all 0's string. Making this space-efficient requires arranging each batch of multiplication gates in neighboring rows.
5) In the final step, the verifier queries the proof oracle on a subset of the columns and verifies if the responses provided by the prover for the code, linear and quadratic tests are consistent with the columns. In the Ligero system, all these tests are the results of row operations on the encoded matrix. Hence the verifier can check the correctness by simply recomputing the row operations on the subset of columns opened by the verifier and checking against the prover's responses. If the tests can be computed by the prover in a space-efficient manner, then the verifier can rely on a similar approach to recompute the responses for the columns in a space-efficient manner.

Space Efficiency by Semantics The main and interesting consequence of the stack-based execution described above is that the entire matrix of wire values is not stored in memory. Indeed, only the rows corresponding to the values in the WASM stack will be maintained and whenever a value is popped, the corresponding encodings are popped and "freed". This is because WASM's operational semantics ensures that popped values will never be referenced again. In slight more detail, we can derive the following garbage collection rules for our system:

Rule-global	Inside a module, global variables are always active
Rule-local	Inside a function frame, local variables are always active
Rule-stack	At any time of execution, values on the stack are active
Rule-free	For any other case, values are inactive and therefore safe to be freed

Those rules are directly mapped to stack operations. By applying the garbage collection rules, the maximum memory consumption is now only proportional to the value stack, rather than the entire matrix. By leveraging state-of-the-art optimization toolchains for high-level programs (Emscripten and LLVM), we can minimize the memory footprint of the generated WASM code.

The Batched Setting. In the batched setting, the prover tries to prove several instances of the same NP relation. Namely, given $\mathcal{R}(\cdot,\cdot)$ and statements $x_1,\ldots,x_\ell$, the prover wants to prove there exist $w_1,\ldots,w_\ell$ such that $\forall_i\ \mathcal{R}(x_i,w_i)=1$. It is known that in the case of the Ligero proof system, one can extract a more efficient system for batched statements. Namely, recalling that we arrange the wire values in a matrix. Then for a batched statement, it holds that the number of wire values of each instance will be the same and we can have the j^{th} row of the matrix carry the j^{th} wires of all the n instances. This arrangement simplifies the linear test because the constraints are uniform throughout the row. An important additional optimization that we identify in this work is that we can eliminate the linear test altogether in the batched setting as these constraints can be implicitly computed. However, eliminating the linear test does not solve the space bottleneck as they still have to be computed and we leverage the WASM semantics to achieve this.

2. Preliminaries

Basic notations. We denote the security parameter by κ. We say that a function $\mu : \mathbb{N} \to \mathbb{N}$ is *negligible* if for every positive polynomial $p(\cdot)$ and all sufficiently large κ's it holds that $\mu(\kappa) < \frac{1}{p(\kappa)}$. We use the abbreviation PPT to denote probabilistic polynomial-time and denote by $[n]$ the set of elements $\{1,\ldots,n\}$ for some $n \in \mathbb{N}$, and by $[a]_i$ the i^{th} element a_i from the set a. For an NP relation $\mathcal{R}$, we denote by $\mathcal{R}_x$ the set of witnesses of x and by $\mathcal{L}_\mathcal{R}$ its associated language. That is, $\mathcal{R}_x = \{w \mid (x,w) \in \mathcal{R}\}$ and $\mathcal{L}_\mathcal{R} = \{x \mid \exists\ w\ s.t.\ (x,w) \in \mathcal{R}\}$.

We assume functions to be represented by a Boolean/arithmetic circuit C (with AND/MULTIPLY, XOR/ADD gates of fan-in 2 and NOT gates), and denote

the size of C by $|\mathrm{C}|$. By default we define the size to include the total number of gates, excluding NOT gates but including input gates. We specify next the definitions of computationally indistinguishable distributions and statistical distance.

3. System Description

In this section, we give a self-contained description of the Ligero system [AHIV17]. For ease of exposition, we will describe it in the Interactive Oracle Proofs (IOP) model [BCS16]. First, we recall some basic notation definitions of the codes used in the Ligero system.

Coding notation. For a code $C \subseteq \Sigma^n$ and vector $v \in \Sigma^n$, denote by $d(v, C)$ the minimal distance of v from C, namely the number of positions in which v differs from the closest codeword in C, and by $\Delta(v, C)$ the set of positions in which v differs from such a closest codeword (in case of ties, take the lexicographically first closest codeword), and by $\Delta(V, C) = \bigcup_{v \in V}\{\Delta(v, C)\}$. We further denote by $d(V, C)$ the minimal distance between a vector set V and a code C, namely $d(V, C) = \min_{v \in V}\{d(v, C)\}$. Our IOP protocol uses Reed-Solomon (RS) codes, defined next.

Definition 3.1 (Reed-Solomon Code)**.** *For positive integers n, k, finite field $\mathbb{F}$, and a vector $\eta = (\eta_1, \ldots, \eta_n) \in \mathbb{F}^n$ of distinct field elements, the code $\mathsf{RS}_{\mathbb{F},n,k,\eta}$ is the $[n, k, n-k+1]$ linear code over $\mathbb{F}$ that consists of all n-tuples $(p(\eta_1), \ldots, p(\eta_n))$ where p is a polynomial of degree $< k$ over $\mathbb{F}$.*

Definition 3.2 (Encoded message)**.** *Let $L = \mathsf{RS}_{\mathbb{F},n,k,\eta}$ be an RS code and $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ be a sequence of distinct elements of $\mathbb{F}$ for $\ell \leq k$. For $u \in L$ we define the message $\mathsf{Decode}_{L,\zeta}(u)$ to be $(p_u(\zeta_1), \ldots, p_u(\zeta_\ell))$, where p_u is the polynomial (of degree $< k$) corresponding to u. For $U \in L^m$ with rows $u^1, \ldots, u^m \in L$, we let $\mathsf{Decode}_{L^m,\zeta}(U)$ be the length-$m\ell$ vector $x = (x_{11}, \ldots, x_{1\ell}, \ldots, x_{m1}, \ldots, x_{m\ell})$ such that $(x_{i1}, \ldots, x_{i\ell}) = \mathsf{Decode}_{L,\zeta}(u^i)$ for $i \in [m]$. Finally, when ζ is clear from the context, we say that U* encodes *x if $x = \mathsf{Decode}_{L^m\zeta}(U)$. All our codes will employ the same $\mathbb{F}, n, \eta$ and we will simply refer the code by RS_k.*

At a very high level, the Ligero IOP protocol proves the satisfiability of an arithmetic circuit C of size s in the following way. The prover arranges (a slightly redundant representation of) the s wire values of C on a satisfying assignment in a matrix, and encodes each row of this matrix using the Reed-Solomon code. The verifier challenges the prover to reveal linear combinations of the entries of the codeword matrix and checks their consistency with t randomly selected columns of this matrix.

3.1. Formal description of the Ligero IOP$(\mathrm{C}, \mathbb{F})$

This section provides a self-contained description of the Ligero IOP for an arithmetic circuit over a (sufficiently large) field $\mathbb{F}$. We remark that the exposition here is a variant of the system described in [AHIV17] that is optimized for a proof length and prover's computation.

- **Input:** The prover $\mathcal{P}$ and the verifier $\mathcal{V}$ share a common input arithmetic circuit $\mathrm{C} : \mathbb{F}^N \to \mathbb{F}$ and input statement x. $\mathcal{P}$ additionally has input $w = (w_1, \ldots, w_N)$ such that $\mathrm{C}(w) = 1$. $\mathcal{P}$ and $\mathcal{V}$ agree on an encoding $\mathsf{RS}_{\mathbb{F},n,k,\eta}$ and ζ. In fact, we will assume there are public algorithms that can generate ζ and η given $\mathbb{F}, n$ and k.
- **Oracle π:** Let m, ℓ be integers such that $m \cdot \ell > N + 3 \cdot s$ where s is the number of multiplication gates in the circuit. For simplicity, we will assume N and s are multiples of ℓ. Then $\mathcal{P}$ generates an extended witness $w_{\mathsf{ext}} \in \mathbb{F}^{m\ell}$ to be w concatenated with the internal wire values, namely $w_1, \ldots, w_N, \alpha_1, \ldots, \alpha_s, \beta_1, \ldots, \beta_s, \gamma_1, \ldots, \gamma_s$ where $(\alpha_i, \beta_i, \gamma_i)$ are the left input, right input and output values of the i^{th} multiplication gate when evaluating $\mathrm{C}(w)$. All affine constraints on the wire values can be encoded via (A, b) where $A \in \mathbb{F}^{m\ell \times m\ell}, b \in \mathbb{F}^{m\ell}$ such that for any w that satisfies C, we have $A \cdot w_{\mathsf{ext}} = b$.

  The prover samples a random codeword $U \in L^m$ where $L = \mathsf{RS}_k$ subject to $w = \mathsf{Decode}_{L^m,\zeta}(U)$ where $\zeta = (\zeta_1, \ldots, \zeta_\ell)$ is a sequence of distinct elements disjoint from $(\eta_1, \ldots, \eta_n)$. $\mathcal{P}$ sets the oracle as $U \in L^m$. Depending on the context, we may view U either as a matrix in $\mathbb{F}^{m \times n}$ in which each row U_i is a purported L-codeword, or as a sequence of n symbols $(U[1], \ldots, U[n])$, $U[j] \in \mathbb{F}^m$.
- **Interactive Protocol:**

1) $\mathcal{V}$ picks randomness:
   a) **[Code test:]** $r_1 \in \mathbb{F}^m$,
   b) **[Linear test:]** $r_2 \in \mathbb{F}^{m\ell}$,
   c) **[Quadratic test:]** $r_3 \in \mathbb{F}^{s/\ell}$.

   and sends (r_1, r_2, r_3) to $\mathcal{P}$.
2) $\mathcal{P}$ responds with $(q_{code}, q_{lin}, q_{quad})$ where:
   a) **[Code test:]** $q_{code} \in \mathbb{F}^n$ is computed as
   $$q_{code} = r_1^T \cdot U, \tag{1}$$
   b) **[Linear test:]** $q_{lin} \in \mathbb{F}^n$ is computed as
   $$q_{lin}[j] = (R_2[j])^T \cdot U[j] \tag{2}$$
   for $j \in [n]$ where R_2 is the unique matrix such that
   $$\mathsf{Decode}_{L_1^m,\zeta}(R_2) = r_2^T \cdot A \tag{3}$$
   where $L_1 = \mathsf{RS}_\ell$.
   c) **[Quadratic test:]** $q_{quad} \in \mathbb{F}^n$ is computed as
   $$q_{quad} = \sum_{i=1}^{s/\ell} (r_3)_i \cdot (U_{\mathsf{left}_i} \odot U_{\mathsf{right}_i} - U_{\mathsf{out}_i}) \tag{4}$$
   Recall that $\mathsf{Decode}_{L^m,\zeta}(U) = (w_1, \ldots, w_N, \alpha_1, \ldots, \alpha_s, \beta_1, \ldots, \beta_s, \gamma_1, \ldots, \gamma_s)$.

Setting $(\mathsf{left}_i, \mathsf{right}_i, \mathsf{out}_i) = (\frac{N}{\ell} + i, \frac{N+s}{\ell} + i, \frac{N+2\cdot s}{\ell} + i)$ we have

$$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{left}_i}) = (\alpha_{\ell\cdot(i-1)+1}, \ldots, \alpha_{\ell\cdot i})$$
$$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{right}_i}) = (\beta_{\ell\cdot(i-1)+1}, \ldots, \beta_{\ell\cdot i})$$
$$\mathsf{Decode}_{L,\zeta}(U_{\mathsf{out}_i}) = (\gamma_{\ell\cdot(i-1)+1}, \ldots, \gamma_{\ell\cdot i})$$

3) $\mathcal{V}$ queries a set $Q \subset [n]$ of t random symbols $U[j]$, $j \in Q$ and accepts iff the following conditions hold:
   a) **[Code test:]** q_{code} is a valid codeword, i.e. $q_{code} \in L$ and for every $j \in Q$, $q_{code}[j] = \sum_{i=1}^{m} (r_1)_i \cdot U_i[j]$.
   b) **[Linear test:]** Let $v = \mathsf{Decode}_{L_2,\zeta}(q_{lin})$ where $L_2 = \mathsf{RS}_{k+\ell}$. Then the verifier checks if the values in v add up to $r_2^T \cdot b$, i.e. $\sum_{i=1}^{\ell} v_i = r_2^T \cdot b$ and for every $j \in Q$, $q_{lin}[j] = (R_2[j])^T \cdot U[j]$ where R_2 is as defined above (noting here that the verifier can locally compute R_2).
   c) **[Quadratic test:]** Let $v' = \mathsf{Decode}_{L_3,\zeta}(q_{quad})$ where $L_3 = \mathsf{RS}_{2\cdot k}$. The verifier checks that every entry of v' is 0 and it holds that $q_{quad}[j] = \sum_{i=1}^{s/\ell} (r_3)_i \cdot (U_{\mathsf{left}_i}[j] \cdot U_{\mathsf{right}_i}[j] - U_{\mathsf{out}_i}[j])$.

The soundness analysis has been argued in [AHIV23] and is formally stated in the following lemma,

Lemma 3.1. *Let e be a positive integer such that $e < d/3$ and suppose that there exists no $\overline{\alpha}$ such that $\mathrm{C}(\overline{\alpha}) = 1$. Then, for any maliciously formed oracle U^* and any malicious prover strategy, the verifier rejects except with at most $(d/|\mathbb{F}|)^{\sigma} + 2/|\mathbb{F}|^{\sigma'} + (1-e/n)^t + 2((e+2k)/n)^t$ probability where σ is the number of times the code test is repeated and σ' is the number of times the linear and quadratic tests are repeated.*

Compiling the Ligero IOP to a SNARK. Compiling the IOP to a SNARK follows a standard compilation [Kil92], [BCS16] using Merkle trees for vector commitments and Fiat-Shamir heuristic. In slightly more detail, the prover uses vector commitments to commit and reveal specific locations of the proof oracle and the random oracle instantiated via a hash function to generate the verifier's random challenges and queries to the oracle and complete the execution (computing its own messages) based on the emulated verifier's messages.

We now analyze the space complexity for a naïve implementation of the Ligero-SNARK prover. We will assume that the prover first evaluates the entire circuit and stores the extended witness w_{ext} in memory which requires space to store $O(s)$ field elements.

- **Generating the root of the Merkle tree committing to the proof oracle:** This proceeds in three steps. (1) Compute the U matrix that is an encoding of w_{ext}, (2) Compute a commitment to $U[j]$ for all $j \in [n]$, and (3) Build a Merkle tree from the n commitments. Since we use a constant distance encoding, the size of U is $O(|w_{\mathsf{ext}}|) = O(s)$ field elements and Reed-Solomon encoding can be done in place without requiring additional memory. For Step 2, with U in memory, the commitments to each column can be computed in a space-efficient manner using Merkle-Damgård hashing. Finally, in Step 3, the Merkle tree on n values can be computed in $O(n)$ space (where recall n is the code length). Overall, this step requires $O(s)$ space.
- **Generating the response to the tests:** With U in memory, computing the response to the code test and the quadratic test does not require any additional space. In the linear test, the prover first generates r_2 using the $m\ell \times m\ell$ matrix A, encodes it to R_2 and computes the result of the linear test by combining R_2 with U. If A is sparse (where each row has a constant number of non-zero elements), it suffices to have $O(m\ell) = O(s)$ memory to compute r_2 and R_2. Overall, this step also requires only $O(s)$ space.
- **Generating U_j for $j \in Q$ with the corresponding Merkle decommitments:** With U matrix in memory, generating the $|Q|$ columns with Merkle decommitments requires no additional space.

Most implementations of ZK-SNARKs utilize $O(s)$-space in memory to implement the prover.

In this work, our goal is to improve space complexity. In particular, we would like the space to be sub-linear in the circuit size. Ideally, we want this to be proportional to the actual space complexity of evaluating the circuit C. In general, identifying the minimal space required to evaluate a circuit C is hard (and relates to a certain pebbling complexity of the graph induced by C). To circumvent this problem, we propose to rely on a computational model with richer semantics than a flat arithmetic circuit, namely WebAssembly or WASM. Next, we describe our prover algorithm by providing the operational semantics of our prover algorithm when interpreting an NP statement expressed as a WASM instance. Then, we argue the space efficiency of our construction.

3.2. Our Approach

We describe the prover and verifier algorithms by providing the operational semantics when interpreting the WASM code. In our implementation we only accommodate a subset of the WASM semantics, specifically, we can only take as input programs that have oblivious control flow. In other words, there cannot be witness-dependent memory access or witness-dependent branching in the program. Interpreting a (limited version) of the WASM code requires the following data structures:

- stack : The main stack where data is pushed before being operated on.
- locals : An array of function arguments and local values used within the scope of the current function frame. A new array is created when a function is called and is deleted upon completion.

Our prover will maintain some additional data structures to build the proof. These are:

- zstack : A stack to maintain the representation of the values corresponding to stack for the proof generation. In the batched setting, this stack will contain the encoding of the values whereas, in the single instance setting, this stack will contain a pointer to the index where the value is stored in the extended witness.
- zlocals : An array to maintain the corresponding representation of the values in locals.
- ctx : A context to prepare the proof.

On a high-level, our approach is to build the U matrix one row at a time and keep only those rows that are carrying "active" values and discard "inactive" rows through garbage collection. This allows us to prepare the proof where the complete set of wire values does not have to be in memory. Then:

1) In Stage 1, the prover will maintain a running hash of each column in the context, updating it when a row is generated. After executing the program, the prover will build the Merkle tree with the aggregated hashes.
2) In Stage 2, the prover will maintain a running aggregate for each of the responses the prover needs to generate for the code, linear and quadratic tests, and will update it as the rows of U are generated.
3) In Stage 3, the prover will simply collect in the context the columns of U that it needs to open while generating U one row at a time.

We consider two instantiations of the Ligetron system. A batched setting where the prover proves several instances of the same (apriori unknown) NP relation and showcases an extremely fast prover and succinct verification. In the second setting, we will consider the more general single-instance or non-batched setting where the prover proves a single instance of (an apriori unknown) NP relation. Next, we describe our batched and non-batched constructions in more detail.

3.2.1. Batched Setting. In the batched setting, the prover wants to prove that $\mathrm{C}(w^i) = 1$ for all $i \in [R]$. In this scenario, the prover can set up w_{ext} so that $\mathsf{Decode}_{L,\zeta}(U_i)$, i.e. the vector encoded in the i^{th} row of the U matrix contains all the i^{th} wire values from R instances. For this, we need to set the "packing factor" of the code $\ell = R$. This way the R values encoded in each row of U_i correspond to the same wire in the R instances. As a consequence, every linear constraint and quadratic constraint applies uniformly over a row. We provide a formal description of our system by providing the operational semantics when we interpret the WASM program.

In the batched setting, each item pushed onto the WASM stack stack will be a vector of values, namely, the value of that variable in the R instances. The zero-knowledge stack zstack will have a 1-1 correspondence with stack where it will maintain the row in U that decodes to the corresponding vector of values in stack. The locals and zlocals array will analogously have a vector of values and their encodings respectively. Next, we describe the execution context ctx.

In stage 1, the execution context is defined as a vector of hash builders. the function $update\_linear$ and $update\_quadratic$ will update the context by appending each column of the encoded polynomial to the corresponding builder. In stage 2, the execution context is defined as a code test row and a quadratic test row. $update\_linear$ will first check the encoding by aggregating the polynomial to the code test row, then check the linear relation by aggregating to the quadratic test row. $update\_quadratic$ will similarly perform the code test first and aggregate to the quadratic test row. In stage 3, the execution context is simply a vector of column-sampled rows. Update functions will sample t columns and append them to the vector.

We provide our operational semantics in Figure 1 for the constant, addition and multiplication operations. The $encode_L(n)$ function produces a random codeword $p \in L$ such that $\mathsf{Decode}_{L,\zeta}(p) = n$. For each of the Stages 1,2 and 3 in our protocol, we provide the context and update functions next:

Stage 1: Here the ctx c contains $h_1, \ldots, h_n$ that are initialized to be empty. The $update\_linear(c, p_1, p_2, p_3)$ and $update\_quadratic(c, p_1, p_2, p_3)$ method updates the hash as follows. It picks random nonces $r_1, \ldots, r_n$ and sets $h_i = SHA256((p_3)_i, r_i)$ for $i \in [n]$.

Stage 2: Here the ctx c contains $q_1, q_2, q_3 \in \mathbb{F}^n$ initialized to all 0s vector and state σ for a pseudo-random generator PRG. The $update\_linear(c, p_1, p_2, p_3)$ method updates q_2 as $q_2 + r \cdot (p_3 - p_1 - p_2)$ where $r \in \mathbb{F}$ is sampled uniformly at random using the PRG. The state of the PRG is updated to σ'. The $update\_quadratic(c, p_1, p_2, p_3)$ method analogously updates q_3 as $q_3 + r \cdot (p_3 - p_1 \odot p_2)$ where $r \in \mathbb{F}$ is sampled uniformly at random using the PRG. The state of the PRG is again updated.

Stage 3: Here the ctx c contains $C_{i_1}, \ldots, C_{i_t}$ and the $update\_linear$ and $update\_quadratic$ methods append the columns by $C_{i_j} = C_{i_j} :: (p_3)_{i_j}$ for $j \in [t]$.

3.2.2. Non-batched. The non-batched execution context is different and significantly harder. We no longer have the uniformity of constraints between the values encoded in each row of U. The main technical challenge is computing the response to the linear test in Stage 2.

In this setting, the prover maintains a set of rows in U that contain the "active" values. More precisely:

- **Linear row r_{tmp}:** The prover maintains a ℓ-element vector r^{val}_{tmp} of values that will be filled with the outputs of linear gates.[2] A corresponding randomness row r^{rand}_{tmp} which is also a vector of ℓ-elements will be maintained. This will be used only in Stage 2 to compute the result of the linear test. In slightly more detail, in the linear test, the prover needs to compute R_2 such that $\mathsf{Decode}_{L_1^m,\zeta}(R_2) = r_2^T \cdot A$. The randomness rows will maintain in r^{rand}_{tmp} the portion of $r_2^T \cdot A$ corresponding to the row r^{val}_{tmp}.

2. For simplicity, we provide our description with a single linear row. Depending on the size of $|globals| + |locals| + |stack|$, we might need multiple rows.

$$\text{State } s ::= \{\mathsf{ctx}\ c;\ \mathsf{locals}\ n^*_{\mathsf{local}};\ \mathsf{zlocals}\ n^*_{\mathsf{zlocal}};\ \mathsf{stack}\ n^*_{\mathsf{stack}}; \mathsf{zstack}\ n^*_{\mathsf{zstack}}; \mathsf{instrs}\ e^*\}$$

$$\frac{p = encode_L(n)}{\{\mathsf{ctx}\ c;\ \mathsf{instrs\ i32.const}\ n\} \mapsto \{\mathsf{ctx}\ c';\ \mathsf{stack}\ n;\ \mathsf{zstack}\ p\}} \text{ (B-Const)}$$

$$\frac{n_3 = n_1\ +\ n_2;\ p_3 = p_1\ +\ p_2;\ c' = update\_linear(c, p_1, p_2, p_3)}{\{\mathsf{ctx}\ c;\ \mathsf{stack}\ n_1, n_2\ \mathsf{zstack}\ p_1, p_2;\ \mathsf{instrs\ i32}.add\} \mapsto \{\mathsf{ctx}\ c';\ \mathsf{stack}\ n_3;\ \mathsf{zstack}\ p_3\}} \text{ (B-Add)}$$

$$\frac{n_3 = n_1\ \odot\ n_2;\ p_3 = encode_L(n_3);\ c' = update\_quadratic(c, p_1, p_2, p_3)}{\{\mathsf{ctx}\ c;\ \mathsf{stack}\ n_1, n_2;\ \mathsf{zstack}\ p_1, p_2;\ \mathsf{instrs\ i32}.mul\} \mapsto \{\mathsf{ctx}\ c';\ \mathsf{stack}\ n_3;\ \mathsf{zstack}\ p_3\}} \text{ (B-Multiply)}$$

Figure 1: Operational Semantics for Batched Setting

- **Quadratic rows r_l, r_r, r_o:** The prover maintains three ℓ-element vectors $(r_l^{val}, r_r^{val}, r_o^{val})$ that will be filled with the inputs and outputs of a multiplication gate. Similar to the linear row, the corresponding randomness rows will be maintained for each of the three rows $(r_l^{rand}, r_r^{rand}, r_o^{rand})$.

The stack will maintain the actual values used by the WASM execution, whereas the zstack will contain a pointer to the values in the corresponding linear and quadratic rows. We describe the semantics of when the rows will be purged and created next. We will rely on reference counting to find out which values in the rows contain active values. We have methods in addition to $update\_linear$ and $update\_quadratic$ in the non-batched setting: $linear\_check$, $push\_ref$, $push$ and $mark\_and\_copy$. The first two methods will update the context when the corresponding rows are fully filled and $linear\_check$ will update the randomness in corresponding rows. The $push$ method will put a value at the end of the specified row and return a reference to it, while the $push\_ref$ method will make a new reference by copying a given reference to the row and building equality constraint. The $mark\_and\_copy$ method requires copying the active values from a fully filled row into a new row and purging the old rows out of memory. We will describe Stage 2 first because this is the most involved whereas the remaining stages use only a subset of the operations performed in Stage 2 during execution.

Stage 2: In this Stage, the ctx c contains $q_{code}, q_{lin}, q_{quad} \in \mathbb{F}^n$ initialized to the all 0's vector and a state σ for a pseudo-random generator (PRG), and proceeds as follows:

- **Addition gate:** When the execution encounters an addition gate, the $push$ method is invoked to push the plain-value calculation result n_3 into next available slot in r_{tmp}^{val}. After that, the $linear\_check$ is invoked to incorporate the linear constraints. To do so, the execution context first identifies the locations of ref_1, ref_2, ref_3 in $r_{tmp}^{val}, r_l^{val}, r_r^{val}, r_o^{val}$. In the corresponding locations $r_{tmp}^{rand}, r_l^{rand}, r_r^{rand}, r_o^{rand}$ the linear constraint is incorporated by first sampling a random field element α and adding $+\alpha$ to the entry in the randomness row corresponding to ref_1 and ref_2 and adding $-\alpha$ corresponding to ref_3. Finally, ref_3 is pushed onto zstack, increasing the reference counting to its location by one.
- **Multiplication gate:** When it encounters a multiplication gate, the execution context first computes the plain-value result of n_3. It will then push the left, right, and output values ref_1, ref_2, n_3 of the multiplication gate to the next available slots in $r_l^{val}, r_r^{val}, r_o^{val}$ respectively. The values are aligned in the same position in each of the three vectors, which is necessary for the quadratic constraint. No matter where ref_1 and ref_2 are pointing, their values always need to be copied because of the alignment requirement. A new reference is created for the locations in r_l^{val}, r_r^{val} where they are copied. A linear constraint is created to ensure the values referenced by ref_1 and ref_2 are copied correctly by the prover context as follows: Two random field elements α_1, α_2 are sampled and in the randomness, rows corresponding to where ref_1 and ref_2's old and new positions we add the values $+\alpha_i$ and $-\alpha_i$ for $i \in \{1, 2\}$ respectively. Reference to n_3 is also created by $push$ and saved on zstack.
- **Linear row is full:** When the linear row r_{tmp}^{val} has no more slots available it triggers the $update\_linear$ method. The $mark\_and\_copy$ method is invoked that creates a new linear row where only active values from the old linear row are copied to the new row. We identify the active values by checking if there are any pointers still referencing the location. Additionally, the randomness rows corresponding to the old and new linear rows are updated with the copy constraint. Then, all the references to the active values in the old row are updated to point to the new row. Next, an encoding of r_{tmp}^{val} and r_{tmp}^{rand} is generated and q_{code} is set to $q_{code} + \alpha \cdot encode(r_{tmp}^{val})$ where α is a random field element, the linear test q_{lin} is set to $q_{lin} + encode(r_{tmp}^{val}) \odot encode(r_{tmp}^{rand})$. Finally, the old linear row $(r_{tmp}^{val}, r_{tmp}^{rand})$ is deleted.
- **Quadratic rows are full:** This occurs when there are no slots available in $r_l^{val}, r_r^{val}, r_o^{val}$. The $update\_quadratic$ is called and the tests are updated as follows. q_{code} is updated as $q_{code} + \alpha_1 \cdot encode(r_l^{val}) + \alpha_2 \cdot encode(r_r^{val}) + \alpha_3 \cdot encode(r_o^{val})$, q_{quad} is updated as $q_{quad} + \alpha \cdot (encode(r_l^{val}) \odot encode(r_r^{val}) -$

$encode(r_o^{val})$) where $\alpha, \alpha_1, \alpha_2, \alpha_3$ are randomly generated field elements, generated using the PRG. Next, similar to the previous event, $mark\_and\_copy$ is triggered for each of the three quadratic rows where all the active values are copied to the linear row. Then the quadratic rows are deleted and new quadratic rows are created.

In Stage 1, we maintain hash aggregates $h_1, \ldots, h_n$ just as in the batched setting and update it whenever a row is purged (i.e. when a linear or quadratic row is full). We do not populate the randomness rows or compute $q_{code}, q_{lin}, q_{quad}$ in this stage.

In Stage 3, we proceed as in Stage 1, and similarly to the batched setting, with the exception that we do not maintain the hash aggregates, but we maintain the column aggregates of the t chosen columns.

In Figure 1 and 4 we describe the semantics for a few of the operations.

Num	::=	i32			
Expression e	::=	i32.const n			
	|	i32.$unop$			
	|	i32.$binop$			
	|	local.get i	|	local.set i
	|	...			
Unop $unop$	::=	eqz	| clz	| ctz	
Binop $binop$	::=	add	| sub	| mul	
	|	div_sx	| shl	| shr_sx	
	|	and	| or	| xor	
	|	rotl	| rotr	| eq	
	|	ne	| lt_sx	| gt_sx	
	|	le_sx	| ge_sx		

Figure 2: A subset of WASM grammar

4. Implementation and Performance

In this section, we provide benchmarks of the Ligetron system and compare them with other post-quantum ZK-SNARKs. Specifically, we compare proof lengths with Aurora, Breakdown, Shockwave, and Orion systems and the running times with Breakdown and Orion systems.

We use Intel® Homomorphic Encryption Acceleration Library [BKS+21] as the primary modular arithmetic and NTT library. It provides an efficient implementation of AVX512 accelerated NTT. On our test machine `c6i.2xlarge`, we utilized the fast AVX512-IFMA accelerated version for less than 50-bit primes. The code, linear and quadratic tests are also accelerated by the fused-multiply-add instruction provided by HEXL. We choose SHA256 provided by OpenSSL 3.0.5 as our hash function. We also provided OpenSSL's SHA3 and LibSodium's Blake2b as an alternative. For the benchmark, we take advantage of the hardware SHA-NI instruction family to achieve the best result. We use OpenMP extensively to parallelize our implementation. The computation of running hash is divided into 256 columns blocks distributed among threads. For non-batched, the encoding of value rows and random rows are parallelized by OpenMP parallel sections.

The experiments were run on AWS `c6i.2xlarge` instances. with a 4-core, 8-thread Intel(R) Xeon(R) Platinum 8375C CPU running at 2.90GHz. The memory capacity is 16GB.We compiled our code with AVX512-DQ and AVX512-IFMA intrinsics and hardware SHA instruction enabled. For each experiment except the browser, we set `OMP_NUM_THREADS=4` and `OMP_PLACES=cores`. Both batch and non-batch browser experiments were running on a MacBook Pro (M1, 2020). The 50 bits prime we used for all experiments is $2^{50} - 2^{38} + 1$.

Security level: We instantiate our system with 128-bit security. Recall from Lemma 3.1, the soundness of our system is

$$(d/|\mathbb{F}|)^{\sigma} + 2/|\mathbb{F}|^{\sigma'} + (1 - e/n)^t + 2((e + 2k)/n)^t$$

Given a circuit size, we first determine the optimum values for e, n, t so that the proof length is minimized and $(1 - e/n)^t + 2((e + 2k)/n)^t$ is smaller than 2^{-129}. Then we set σ and σ' so that $d/|\mathbb{F}|^{\sigma} + 2/|\mathbb{F}|^{\sigma'}$ is smaller than 2^{-129} to get an overall soundness of 2^{-128}.

Given the need to interpret the WASM code three times with different execution contexts, the implementation is organized as a double-dispatch interpreter. One dispatch is based on each WASM instruction and the other one is based on the execution context. We defined three separate prover execution contexts for each stage and a verifier context, both derived from the basic WASM evaluation context. The whole program works as follows: Upon initialization, the prover loads the WASM file and parses it as an AST. For parsing the binary format WASM code, we utilized the official WASM binary toolkit WABT. We also perform an AST-to-AST translation pass to make it suitable for the interpreter. Once the AST is parsed, a WASM module is instantiated and initialized with a set of external values and imported functions. The prover then starts executing the proof system by calling $call$ instruction with the start function $\_start$, repeating for the corresponding context.

4.1. Proof Length

In Table 1 we compare our proof lengths with the `libiop` implementation of Ligero, Aurora, Breakdown, Shockwave, and Orion. The data for the previous works was obtained from [GLS+21a] where all schemes were set to 128-bit security. The new analysis of the Ligero system yields proof lengths that are significantly better than the other IOP-based implementations. We can see from the figure that Ligetron achieves the best proof length up to 2^{20} gates. We believe our system will have competitive proof

$$\frac{\begin{array}{c} n_3 = n_1 \ \vee \ n_2 \\ arr_1 = bit\_decompose(n_1);\ arr_2 = bit\_decompose(n_2) \\ p = sum((p_a + p_b - p_a p_b) p_{factor} \mid i \leftarrow 0 \ldots |n_3|,\ p_a \leftarrow encode(arr_1[i]),\ p_b \leftarrow encode(arr_2[i]),\ p_{factor} \leftarrow encode(\mathbf{2^i})) \\ c_1 = update\_linear(c, p_a, p_b, p_a + p_b);\ c_2 = update\_quadratic(c_1, p_a, p_b, p_a p_b) \\ c_3 = update\_linear(c_2, p_a + p_b - p_a p_b, p_a p_b, p_a + p_b) \end{array}}{\{\mathsf{ctx}\ c;\ \mathsf{stack}\ n_1, n_2;\ \mathsf{zstack}\ p_1, p_2;\ \mathsf{instrs}\ \mathsf{i32}.or\} \mapsto \{\mathsf{ctx}\ c_3;\ \mathsf{stack}\ n_3;\ \mathsf{zstack}\ p\}}\ \text{(B-BitOr)}$$

$$\frac{\begin{array}{c} n_3 = n_1 \ == \ n_2;\ p_{one} = encode(\mathbf{1}) \\ diff = bit\_decompose(n_2 \ - \ n_1),\ p_{acc} = p_{one} \\ p = sum((p_{one} - p_{diff}) p_{acc} \mid i \leftarrow 0 \ldots |n_3|,\ p_{diff} \leftarrow encode(diff[i])) \\ c' = update\_quadratic(update\_linear(c, p_{one} - p_{diff}, p_{diff}, p_{one}), p_{acc}, p_{one} - p_{diff}, p_{acc}(p_{one} - p_{diff})) \end{array}}{\{\mathsf{ctx}\ c;\ \mathsf{stack}\ n_1, n_2;\ \mathsf{zstack}\ p_1, p_2;\ \mathsf{instrs}\ \mathsf{i32}.eq\} \mapsto \{\mathsf{ctx}\ c';\ \mathsf{stack}\ n_3;\ \mathsf{zstack}\ p\}}\ \text{(B-Equal)}$$

$$\frac{}{\{\mathsf{locals}\ l;\ \mathsf{zlocals}\ p_{locals};\ \mathsf{instrs}\ \mathsf{local.get}\ i\} \mapsto \{\mathsf{locals}\ l;\ \mathsf{zlocals}\ p_{locals};\ \mathsf{stack}\ l[i];\ \mathsf{zstack}\ p_{locals}[i]\}}\ \text{(B-LocalGet)}$$

$$\frac{}{\{\mathsf{locals}\ l;\ \mathsf{zlocals}\ p_{locals}; \mathsf{stack}\ n; \mathsf{zstack}\ p_n;\ \mathsf{instrs}\ \mathsf{local.set}\ i\} \mapsto \{\mathsf{locals}\ l\ \mathsf{with}\ l[i] = n;\ \mathsf{zlocals}\ p_{locals}\ \mathsf{with}\ p_{locals}[i] = p_n\}}\ \text{(B-LocalSet)}$$

$$\frac{\begin{array}{c} n_3 = n_1 \ \oplus \ n_2 \\ arr_1 = bit\_decompose(ref_1);\ arr_2 = bit\_decompose(ref_2) \\ ref = sum((arr_1[i] + arr_2[i] - arr_1[i] arr_2[i]) ref_{fact} \mid i \leftarrow 0 \ldots |n_3|,\ ref_{fact} \leftarrow push(r_{tmp}, 2^i)) \end{array}}{\{\mathsf{rows}\ r_{tmp};\ \mathsf{stack}\ n_1, n_2;\ \mathsf{zstack}\ ref_1, ref_2;\ \mathsf{instrs}\ \mathsf{i32}.or\} \mapsto \{\mathsf{rows}\ r_{tmp};\ \mathsf{stack}\ n_3;\ \mathsf{zstack}\ ref\}}\ \text{(NB-BitOr)}$$

$$\frac{\begin{array}{c} n_3 = n_1 \ == \ n_2 \\ refs_{diff} = bit\_decompose(ref_2 - ref_1) \\ ref = product((1 - ref_d) \mid i \leftarrow 0 \ldots |n_3|,\ ref_d \leftarrow refs_{diff}[i]) \end{array}}{\{\mathsf{stack}\ n_1, n_2;\ \mathsf{zstack}\ ref_1, ref_2;\ \mathsf{instrs}\ \mathsf{i32}.eq\} \mapsto \{\mathsf{stack}\ n_3;\ \mathsf{zstack}\ ref\}}\ \text{(NB-Equal)}$$

$$\frac{}{\{\mathsf{locals}\ l;\ \mathsf{zlocals}\ ref_l;\ \mathsf{instrs}\ \mathsf{local.get}\ i\} \mapsto \{\mathsf{locals}\ l;\ \mathsf{zlocals}\ ref_l;\ \mathsf{stack}\ l[i];\ \mathsf{zstack}\ ref_l[i]\}}\ \text{(NB-LocalGet)}$$

$$\frac{}{\{\mathsf{locals}\ l;\ \mathsf{zlocals}\ ref_l; \mathsf{stack}\ n; \mathsf{zstack}\ ref_n;\ \mathsf{instrs}\ \mathsf{local.set}\ i\} \mapsto \{\mathsf{locals}\ l\ \mathsf{with}\ l[i] = n;\ \mathsf{zlocals}\ ref_l\ \mathsf{with}\ ref_l[i] = ref_n\}}\ \text{(NB-LocalSet)}$$

Figure 3: More Operational Semantics

sizes up to a billion gates. We remark that the main improvement of Ligetron's proof length from the original Ligero work is the improved soundness analysis from [AHIV23] that in turn relied on improvements in Reed-Solomon code testing.

4.2. Single Instance or Non-Batched Setting

In this section, we consider the case where the proof of a single instance of an NP relation is generated. We considered four different examples to benchmark our system and compare it with prior work. In each case, we coded the relation in C/C++ and then cross-compiled the program to WASM using emscripten. We used this WASM file as input for the prover and verifier algorithm.

We measured the end-to-end performance of our system. The input to our system is a "succinct" WASM code that represents the functionality. Both our prover and verifier generate the gates/constraints of the corresponding circuit on-the-fly and this is included in the overall running time. In contrast, the systems we compare with use R1CS as an input which is essentially a "flattened" arithmetic circuit corresponding to the NP relation and, in particular, the timing excludes circuit generation cost.

Next, we describe our examples.

1. **SHA Circuit** The prover demonstrates that it knows a pre-image of a certain length under the SHA256 hash function. More precisely, the instance is a 256-bit string h, and integer n and the witness is a n-bit string x. The relation checks that x is of length n and $SHA256(x) = h$. In Figures 6(a)-6(c) we give the prover time, verifier time and peak memory of the prover and verifier for different values of n.
2. **Random Circuit** We coded a random circuit via a C-program, compiled to WASM and generated a proof interpreting the WASM file as the NP-relation. We prove the prover, verifier times and peak memory as for the SHA in Figures 6[d]-6[f]. In this set of experiments, we compare our performance against prior works Ligero, Brakedown and Orion.
3. **AES Circuit** Similar to SHA example, the prover shows that it knows a key k and message m such that $AES128(k, m) = c$ for an instance c. In this benchmark, we compare Ligetron's performance with Ligero. See Figures 6[g]-6[i].
4. **Combination Circuit** Finally, we benchmarked our system on a circuit that combines multiple primitives. We considered a scenario where the prover and verifier

$$\text{State } s ::= \ \{\ \ldots,\ \mathsf{rows}\ r_{tmp},\ r_l,\ r_r,\ r_o\ \}$$

$$\frac{\begin{array}{c} r' = makeRow();\ mark\_and\_copy(r_{tmp}, r') \\ c' = update\_linear(c, encode(r_{tmp})) \end{array}}{\{\mathsf{ctx}\ c;\ \mathsf{rows}\ r_{tmp}\} \mapsto \{\mathsf{ctx}\ c';\ \mathsf{rows}\ r'\}}\ \text{(GC-Tmp)}$$

$$\frac{\begin{array}{c} r'_l = makeRow();\ mark\_and\_copy(r_l, r_{tmp}) \\ r'_r = makeRow();\ mark\_and\_copy(r_r, r_{tmp}) \\ r'_o = makeRow();\ mark\_and\_copy(r_o, r_{tmp}) \\ c' = update\_quadratic(c, encode(r_l), encode(r_r), encode(r_o)) \end{array}}{\{\mathsf{ctx}\ c;\ \mathsf{rows}\ r_l\ r_r\ r_o\} \mapsto \{\mathsf{ctx}\ c';\ \mathsf{rows}\ r'_l\ r'_r\ r'_o\}}\ \text{(GC-Quadratic)}$$

$$\frac{ref = push(r_{tmp}, n)}{\{\mathsf{rows}\ r_{tmp};\ \mathsf{instrs}\ \mathsf{i32.const}\ n\} \mapsto \{\mathsf{rows}\ r_{tmp};\ \mathsf{stack}\ n;\ \mathsf{zstack}\ ref\}}\ \text{(NB-Const)}$$

$$\frac{\begin{array}{c} n_3 = n_1\ +\ n_2 \\ ref_3 = push(r_{tmp}, n_3) \\ linear\_check(ref_1, ref_2, ref_3) \end{array}}{\{\mathsf{rows}\ r_{tmp};\ \mathsf{stack}\ n_1, n_2\ \mathsf{zstack}\ ref_1, ref_2;\ \mathsf{instrs}\ \mathsf{i32}.add\} \mapsto \{\mathsf{stack}\ n_3;\ \mathsf{zstack}\ ref_3\}}\ \text{(NB-Add)}$$

$$\frac{\begin{array}{c} n_3 = n_1\ *\ n_2 \\ push\_ref(r_l, ref_1);\ push\_ref(r_r, ref_2);\ ref_3 = push(r_o, n_3) \end{array}}{\begin{array}{c}\{\mathsf{rows}\ r_l,\ r_r,\ r_o;\ \mathsf{stack}\ n_1, n_2;\ \mathsf{zstack}\ ref_1, ref_2;\ \mathsf{instrs}\ \mathsf{i32}.mul\} \\ \mapsto \{\mathsf{rows}\ r_l,\ r_r,\ r_o;\ \mathsf{stack}\ n_3;\ \mathsf{zstack}\ ref_3\}\end{array}}\ \text{(NB-Multiply)}$$

Figure 4: Operational Semantics for non-Batched Setting

Circuit Size	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
Ligero [AHIV17]	546	628	1,076	1,169	2,100	3,169	5,788	5,662	10,527	10,736	19,828
Aurora [BCR+19]	447	510	610	717	810	931	1,069	1,179	1,315	1,473	1,603
Brakedown(128-bit) [GLS+21a]	1,279	1,597	1,974	2,200	2,710	3,165	3,926	4,824	6,122	7,899	10,230
Shockwave(128-bit) [GLS+21a]	72	95	122	160	210	284	386	523	721	990	1,384
Orion					1064	1064	1151	1248	1354	1461	1573
Ligetron-128 (here)	48	56	71	87	103	135	177	229	320	417	602

TABLE 1: Comparison of proof lengths (in KB) between different IOP-based ZK-SNARKs.

hold a hash h, a public word w, constants x, y and the prover privately holds a database of words D. We generate a proof that proves that there exists D such that $SHA256(D) = h$ and there are at least x occurrences of words whose edit distance from w is at most y. This set of experiments showcases our scalability and how we can easily code an application by coding it in a high-level language.

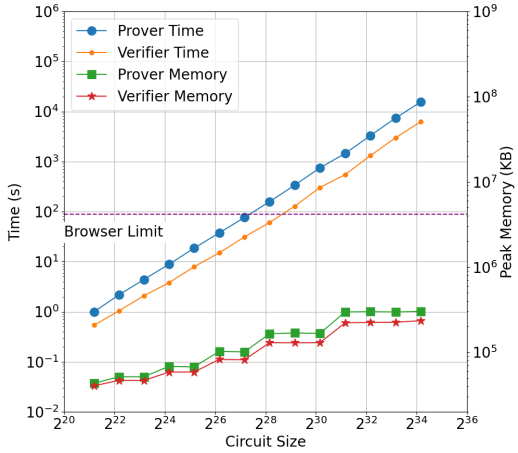

Figure 5: Prover and Verifier running times and peak memory for various sizes of the combination circuit.

We observe that our prover and verifier times are fairly consistent across all examples. We ran our system up to 20 billion ($2^{34} \approx 1.7 \cdot 10^{10}$) gates for the combination circuit and our prover runs at roughly 500ns/gate and our verifier at 250ns/gate.

Comparison with Brakedown and Orion. We compare our performance with Brakedown and Orion when benchmarking on a random circuit. The Brakedown and Orion systems provide benchmarks for a random R1CS instance of different sizes. To benchmark Ligetron on a random circuit, we coded a random circuit in C and compiled it to WASM. In Figures 6[d] and 6[e] we can see that the Ligetron prover is 5-6x faster than Brakedown and Orion while the Ligetron verifier is 1.5-2x faster. On the memory front, Ligetron performs better than Brakedown and Orion, however, the memory consumption of Ligetron increases with the size of the circuit. This is expected as a random circuit is unstructured. We were able to run Ligetron upto 2^{25} gates, while the Brakedown and Orion systems ran out of memory at 2^{22} and 2^{23} gates. For structured circuits such as AES, SHA and our combination circuit, we can see that Ligetron's memory consumption grows fairly slowly. The increase in memory consumption is related to the packing factor we chose to optimize the proof length. It is possible to tradeoff the proof length to get optimal space-efficiency. The largest circuit we tested our system was 17 Billion gates (on the combination circuit). Our system should be able to handle arbitrarily large circuits as long as the space-consumption to naively evaluate the WASM code is bounded.

Comparison with Ligero. The base IOP used in Ligetron is the same as in Ligero. For a fair comparison with Ligero (i.e. having all performance enhancements from Intel-intrinsics AVX and SHA kept the same), we benchmark the Ligetron prover and verifier with *garbage collection* switched off. This serves as an approximation of the Ligero system as the entire extended witness and its encoding U are kept in memory. We compare Ligetron's performance with the Ligero-approximation on a random circuit and a structured circuit, AES. Ligetron's prover and verifier times are around 1.2x faster than Ligero, however, the memory consumption is far better. Specifically on the AES circuit, the Ligero-approximation ran out of memory at 2^7 AES calls, whereas, Ligetron's memory remains fairly low.

Performance on a browser. We cross-compiled our prover and verifier algorithms to WASM and executed our code from a browser. In Table 2, we provide our prover and verifier running times for the combination NP relation for various sizes. We see that there is a 7x slowdown on the prover and a 6x slowdown on the verifier. On c6i.2xlarge, we gain from both intrinsics and mild parallelism which are not available from the browser.

4.3. The Batched Setting

We also benchmarked our system in the batched setting where the prover proves several instances of the same NP relation. In Figure 7 we present the prover and verifier running times along with the peak memory usage. As can be seen, our speeds in the batched setting are orders of magnitude faster than in the non-batched setting. Our prover runs at 65ns/gate while our verifier runs at 3ns/gate. This stands competitive against all current concretely efficient zero-knowledge systems (non-interactive and interactive). We also present in Table 3 the running times of our prover and verifier on a browser.

Circuit size	Prover Time	Verifier Time
37,701,960	3.23 μs/g	1.42 μs/g
150,645,296	3.32 μs/g	1.47 μs/g
602,368,646	3.43 μs/g	1.53 μs/g
1,204,606,598	3.44 μs/g	1.54 μs/g

TABLE 2: Prover and Verifier times on the Firefox browser 110.0.1 (64-bit) run on a MacBook Pro (M1, 2020). μs/g is microseconds per gate.

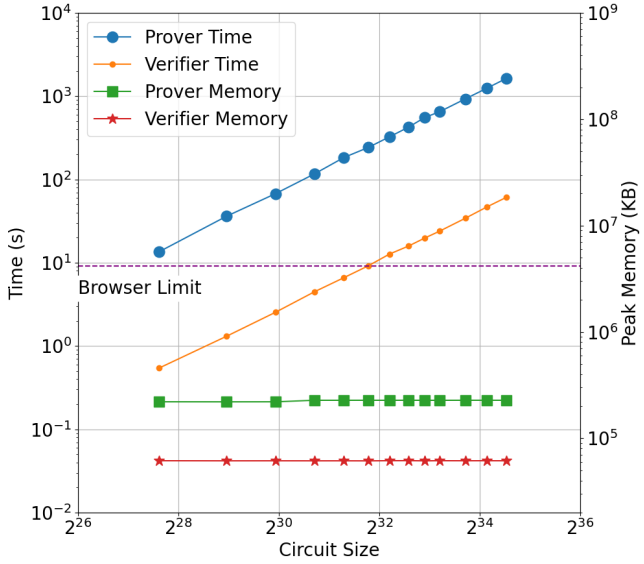

Figure 7: Prover and Verifier running times and peak memory for various sizes of the edit distance computation in the batched setting.

String length	Cons./ batch	Batch size	Prover Time	Verifier Time
10	40,804	1024	1.19 μs/g	11.2 ns/g
30	334,820	1024	1.22 μs/g	14.1 ns/g
40	589,676	1024	1.21 μs/g	12.9 ns/g
50	915,684	1024	1.17 μs/g	11.8 ns/g

TABLE 3: Prover and verifier running times on the Brave browser run on a MacBook Pro (M1, 2020). μs/g is microseconds per gate and ns/g is nanoseconds per gate.

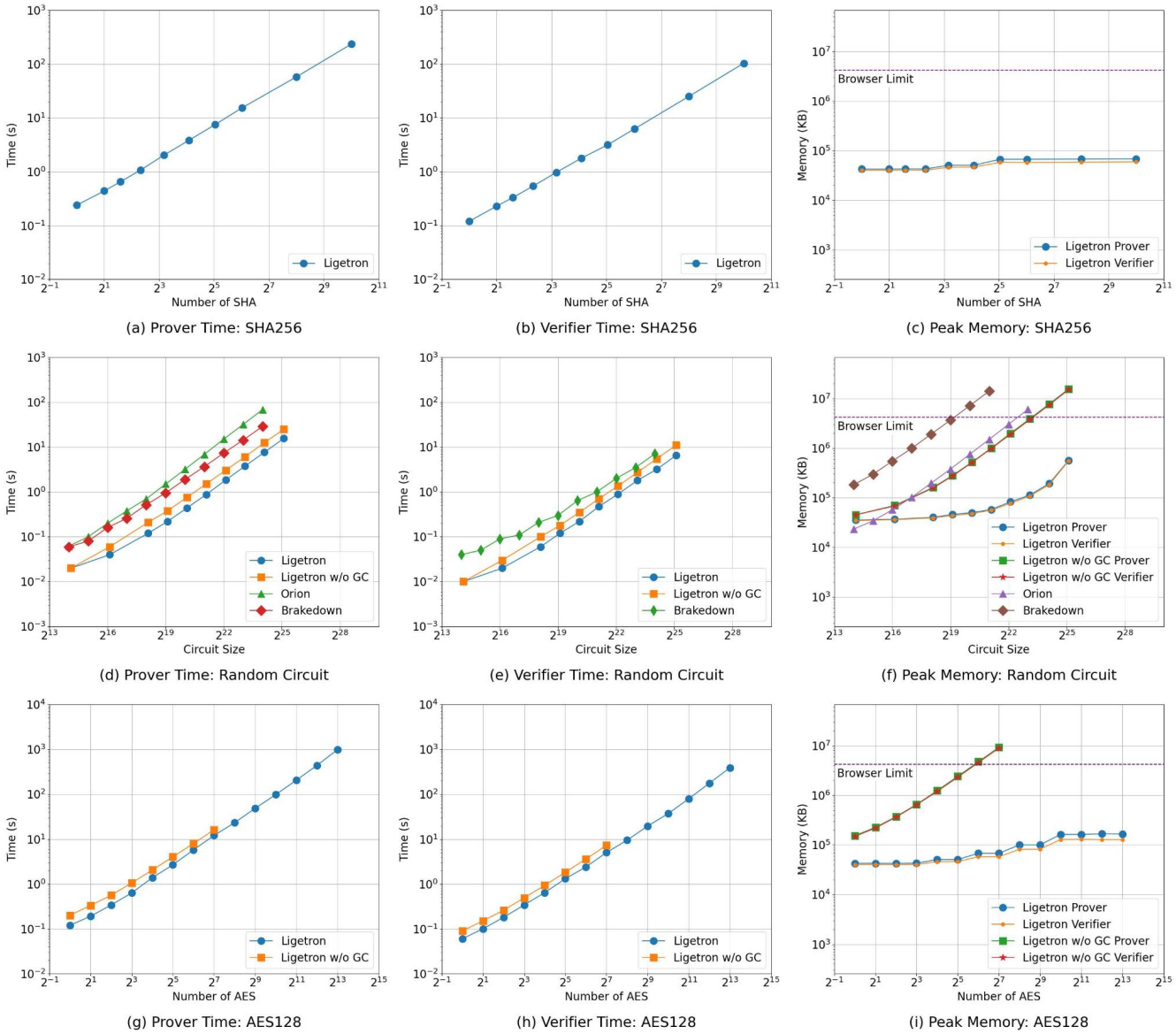

Figure 6: Benchmarks for SHA, Random Circuit and AES

5. Conclusions

We introduced a novel compilation of NP statements to a ZK-SNARK based on the Ligero system. Our system is lightweight and scalable. Given that we can cross-compile our algorithms to WASM and run it from a browser, it will make it easy to deploy ZK applications on a Web Application with state-of-the-art performance. Another advantage of compiling the verification algorithm to WASM is that we could feed the verification circuit as input NP relation to the prover. Such a composition will be useful for incrementally verifiable computation.

Which applications are suitable for Ligetron. Our benchmarks indicate that the performance of Ligetron on simple programs is consistent. The benchmarks used in prior work fall into two categories: structured simple programs such as repeated SHA/AES computations, matrix multiplication and random circuits (R1CS). We believe Ligetron will be competitive on most if not all structured programs. These include dynamic programming algorithms, neural network computations, blockchain rollups. On unstructured circuits, Ligetron could still be competitive as it can take advantage of the compiler optimizations in the LLVM-compiler (emscripten) and garbage collection on intermediate variables generated as part of various operations (eg, bit decomposition).

Our current implementation can only input WASM code that has oblivious control flow. This does not restrict the class of statements we can use with our system as any R1CS

can be translated to a WASM code where the control flow will be oblivious. Such a transformation, however, will take advantage of the space-efficiency of Ligetron. An important next step will be to be able to take any WASM code as input even those with non-oblivious control flow (eg, RAM, secret variable branching). Finally, Ligetron's main feature is scalability via on-the-fly circuit/constraint generation which is suitable for applications where the actual size of the instance can be chosen at run time.

6. Acknowledgements

We thank the anonymous reviewers for their insightful comments and suggestions. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). The authors are supported by DARPA under Contract No. HR001120C0087. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

References

[AHIV17] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *CCS*, pages 2087–2104, 2017.

[AHIV23] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: lightweight sublinear arguments without a trusted setup. *Designs, Codes and Cryptography*, July 2023.

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3):501–555, 1998.

[AS98] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of NP. *J. ACM*, 45(1):70–122, 1998.

[Bab85] László Babai. Trading group theory for randomness. In *STOC*, pages 421–429, 1985.

[BBB+18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 315–334. IEEE Computer Society, 2018.

[BBHR19] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO*, pages 701–732, 2019.

[BBHV22] Laasya Bangalore, Rishabh Bhadauria, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. On black-box constructions of time and space efficient sublinear arguments. *TCC 2022. To Appear*, 2022.

[BCL20] Jonathan Bootle, Alessandro Chiesa, and Siqi Liu. Zero-knowledge succinct arguments with a linear-time prover. *IACR Cryptol. ePrint Arch.*, page 1527, 2020.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. Recursive proof composition from accumulation schemes. In *TCC*, pages 1–18, 2020.

[BCR+19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for R1CS. In *EUROCRYPT*, pages 103–128, 2019.

[BCS16] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *TCC*, pages 31–60, 2016.

[BFH+20] Rishabh Bhadauria, Zhiyong Fang, Carmit Hazay, Muthuramakrishnan Venkitasubramaniam, Tiancheng Xie, and Yupeng Zhang. Ligero++: A new optimized sublinear IOP. In *CCS*, pages 2025–2038, 2020.

[BFLS91] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *STOC*, pages 21–31, 1991.

[BGH19] Sean Bowe, Jack Grigg, and Daira Hopwood. Halo: Recursive proof composition without a trusted setup. *IACR Cryptol. ePrint Arch.*, page 1021, 2019.

[BHR+20] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Public-coin zero-knowledge arguments with (almost) minimal time and space overheads. In *TCC*, pages 168–197, 2020.

[BHR+21] Alexander R. Block, Justin Holmgren, Alon Rosen, Ron D. Rothblum, and Pratik Soni. Time- and space-efficient arguments from groups of unknown order. In *CRYPTO*, pages 123–152, 2021.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). https://github.com/intel/hexl, september 2021.

[BMRS21] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. Mac'n'cheese: Zero-knowledge proofs for boolean and arithmetic circuits with nested disjunctions. In *CRYPTO*, pages 92–122, 2021.

[CDMP05] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In *CRYPTO*, pages 430–448, 2005.

[CHM+20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In *EUROCRYPT*, pages 738–768, 2020.

[COS20] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In *EUROCRYPT*, pages 769–793, 2020.

[GLS+21a] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and post-quantum snarks for R1CS. *IACR Cryptol. ePrint Arch.*, page 1043, 2021.

[GLS+21b] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and post-quantum snarks for R1CS. *IACR Cryptol. ePrint Arch.*, page 1043, 2021.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, pages 291–304, 1985.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, page 953, 2019.

[Ish20] Yuval Ishai. Zero-knowledge proofs from information-theoretic proof systems, 2020. https://zkproof.org/2020/08/12/information-theoretic-proof-systems.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 723–732, 1992.

[KMP20] Abhiram Kothapalli, Elisaweta Masserova, and Bryan Parno. A direct construction for asymptotically optimal zksnarks. *IACR Cryptol. ePrint Arch.*, page 1318, 2020.

[KR08] Yael Tauman Kalai and Ran Raz. Interactive PCP. In *ICALP*, pages 536–547, 2008.

[LSTW21] Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Linear-time zero-knowledge snarks for R1CS. *IACR Cryptol. ePrint Arch.*, page 30, 2021.

[Mat] https://github.com/matter-labs/awesome-zero-knowledge-proofs.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *CCS*, pages 2111–2128, 2019.

[Mer89] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.

[QED] https://qed-it.github.io/zkinterface-wasm-demo/.

[ris] Risc Zero. The General Purpose Zero-Knowledge VM. https://www.risczero.com/.

[RRR16] Omer Reingold, Guy N. Rothblum, and Ron D. Rothblum. Constant-round interactive proofs for delegating computation. In *STOC*, pages 49–62, 2016.

[RS60] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[Set20] Srinath T. V. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO*, pages 704–737, 2020.

[SL20] Srinath T. V. Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zksnarks. *IACR Cryptol. ePrint Arch.*, page 1275, 2020.

[Tha21] Justin Thaler. Proofs, arguments, and zero-knowledge, 2021. https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.html.

[WTS+18] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *S&P*, pages 926–943, 2018.

[WYKW21] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. Wolverine: Fast, scalable, and communication-efficient zero-knowledge proofs for boolean and arithmetic circuits. In *SP*, pages 1074–1091, 2021.

[WYY+22] Chenkai Weng, Kang Yang, Zhaomin Yang, Xiang Xie, and Xiao Wang. Antman: Interactive zero-knowledge proofs with sublinear communication. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2901–2914. ACM, 2022.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*, volume 13510 of *Lecture Notes in Computer Science*, pages 299–328. Springer, 2022.

[XZZ+19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO*, pages 733–764, 2019.

[YSWW21] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2986–3001. ACM, 2021.

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *S&P*, pages 859–876. IEEE, 2020.

Appendix A. Basic Definitions

A.1. Zero-Knowledge Arguments

We denote by $\langle A(w), B(z)\rangle(x)$ the random variable representing the (local) output of machine B when interacting with machine A on common input x, when the random-input to each machine is uniformly and independently chosen, and A (resp., B) has auxiliary input w (resp., z).

Definition A.1 (Interactive argument system)**.** *A pair of* PPT *interactive machines $\langle \mathcal{P}, \mathcal{V}\rangle$ is called an* interactive proof system for a language $\mathcal{L}$ *if there exists a negligible function* negl *such that the following two conditions hold:*

1) COMPLETENESS: *For every $x \in \mathcal{L}$ there exists a string w such that for every $z \in \{0,1\}^*$,*

$$\Pr[\langle \mathcal{P}(w), \mathcal{V}(z)\rangle(x) = 1] \geq 1 - \mathsf{negl}(|x|).$$

2) SOUNDNESS: *For every $x \notin \mathcal{L}$, every interactive* PPT *machine $\mathcal{P}^*$, and every $w, z \in \{0,1\}^*$*

$$\Pr[\langle \mathcal{P}^*(w), \mathcal{V}(z)\rangle(x) = 1] \leq \mathsf{negl}(|x|).$$

Definition A.2 (Zero-knowledge)**.** *Let $\langle \mathcal{P}, \mathcal{V}\rangle$ be an interactive proof system for some language $\mathcal{L}$. We say that $\langle \mathcal{P}, \mathcal{V}\rangle$ is* computational zero-knowledge with respect to an auxiliary input *if for every* PPT *interactive machine $\mathcal{V}^*$ there exists a* PPT *algorithm $\mathcal{S}$, running in time polynomial in the length of its first input, such that*

$$\{\langle \mathcal{P}(w), \mathcal{V}^*(z)\rangle(x)\}_{x\in\mathcal{L}, w\in\mathcal{R}_x, z\in\{0,1\}^*} \overset{c}{\approx} \{\langle \mathcal{S}\rangle(x,z)\}_{x\in\mathcal{L}, z\in\{0,1\}^*}$$

(when the distinguishing gap is considered as a function of $|x|$). Specifically, the left term denote the output of $\mathcal{V}^*$ after it interacts with $\mathcal{P}$ on common input x whereas, the right term denote the output of $\mathcal{S}$ on x.

A.2. Interactive Oracle Proofs

Interactive Oracle Proofs (IOP) [BCS16], [RRR16] is a type of proof system that combines the aspects of Interactive Proofs (IP) [Bab85], [GMR85] along with Probabilistic Checkable Proofs (PCP) [BFLS91], [AS98], [ALM+98] as well generalizes Interactive PCPs (IPCP) [KR08]. In this model, like the PCP model, the verifier does not need to read the whole proof and instead can query the proof at

some random locations while similarly to the IP model, the prover and verifier interact over several rounds.

A k-round IOP has k rounds of interaction. In the i^{th} round of interaction, the verifier sends a uniform public message m_i to the prover and the prover generates π_i. After running k rounds of interaction, the verifier makes some queries to the proofs via oracle access and will either accept it or reject it.

Definition A.3. *Let $\mathcal{R}(x,\omega)$ be an NP relation corresponding to an NP language $\mathcal{L}$. An IOP system for a relation $\mathcal{R}$ with round complexity k and soundness ϵ is a pair of PPT algorithms $(\mathcal{P},\mathcal{V})$ if it satisfies the following properties:*

- SYNTAX: *On common input x and prover input ω, $\mathcal{P}$ and $\mathcal{V}$ run an interactive protocol of k rounds. In each round i, $\mathcal{V}$ sends a message m_i and $\mathcal{P}$ generates π_i. Here the verifier has oracle access to $\{\pi_1,\pi_2,\ldots,\pi_k\}$. We can express $\pi=(\pi_1,\pi_2,\ldots,\pi_k)$. Based on the queries from these oracles, $\mathcal{V}$ accepts or rejects.*
- COMPLETENESS: *If $(x,\omega)\in\mathcal{R}$ then,*

$$\Pr[(\mathcal{P}(x,\omega),\mathcal{V}^{\pi}(x))=1]=1$$

- SOUNDNESS: *For every $x\notin\mathcal{L}$, every unbounded algorithm $\mathcal{P}^*$ and proof $\tilde{\pi}$*

$$\Pr[(\mathcal{P}^*,\mathcal{V}^{\tilde{\pi}})=1]\leq \mathsf{negl}(\lambda)$$

The notion of IOP can be extended to provide zero-knowledge property as well. Next we define the definition of zero-knowledge IOP.

Definition A.4. *Let $\langle\mathcal{P},\mathcal{V}\rangle$ be an IOP for $\mathcal{R}$. We say that $\langle\mathcal{P},\mathcal{V}\rangle$ is a (honest verifier) zero-knowledge IOP (or ZKIOP for short) if there exists a PPT simulator $\mathcal{S}$, such that for any $(x,\omega)\in\mathcal{R}$, the output of $\mathcal{S}(x)$ is distributed identically to the view of $\mathcal{V}$ in the interaction $(\mathcal{P}(x,\omega),\mathcal{V}^{\pi}(x))$.*

A.3. Collision-Resistant Hashing and Merkle Trees

Let $\{\mathcal{H}_\kappa\}_{\kappa\in\mathbb{N}}=\{H:\{0,1\}^{p(\kappa)}\to\{0,1\}^{p'(\kappa)}\}_\kappa$ be a family of hash functions, where $p(\cdot)$ and $p'(\cdot)$ are polynomials so that $p'(\kappa)\leq p(\kappa)$ for sufficiently large $\kappa\in\mathbb{N}$. For a hash function $H\leftarrow\mathcal{H}_\kappa$ a Merkle hash tree [Mer89] is a data structure that allows to commit to $\ell=2^d$ messages by a single hash value h such that revealing any message requires only to reveal $O(d)$ hash values.

A Merkle hash tree is represented by a binary tree of depth d where the ℓ messages $m_1,\ldots,m_\ell$ are assigned to the leaves of the tree; the values assigned to the internal nodes are computed using the underlying hash function H that is applied on the values assigned to the children, whereas the value h that commits to $m_1,\ldots,m_\ell$ is assigned to the root of the tree. To open the commitment to a message m_i, one reveals m_i together with all the values assigned to nodes on the path from the root to m_i, and the values assigned to the siblings of these nodes. We denote the algorithm of committing to ℓ messages $m_1,\ldots,m_\ell$ by $h:=\mathsf{Commit}_{\mathsf{M}}(m_1,\ldots,m_\ell)$ and the opening of m_i by $(m_i,\mathrm{path}(i)):=\mathsf{Open}_{\mathsf{M}}(h,i)$. Verifying the opening of m_i is carried out by essentially recomputing the entire path bottom-up and comparing the final outcome (i.e., the root) to the value given at the commitment phase.

The binding property of a Merkle hash tree is due to collision-resistance. Intuitively, this says that it is infeasible to efficiently find a pair (x,x') so that $H(x)=H(x')$, where $H\leftarrow\mathcal{H}_\kappa$ for sufficiently large κ. In fact, one can show that collision-resistance of $\{\mathcal{H}_\kappa\}_{\kappa\in\mathbb{N}}$ carries over to the Merkle hashing. Formally, we say that a family of hash functions $\{\mathcal{H}_\kappa\}_\kappa$ is collision-resistant if for any PPT adversary $\mathcal{A}$ the following experiment outputs 1 with probability $\mathsf{negl}(\kappa)$: (i) A hash function H is sampled from $\mathcal{H}_\kappa$; (ii) The adversary $\mathcal{A}$ is given H and outputs x,x'; (iii) The experiment outputs 1 if and only if $x\neq x'$ and $H(x)=H(x')$.

In the random oracle model, Merkle tree can be computed by replacing the function H with a random oracle ρ where statistical binding follows due to the hardness of finding a collision in this model. We denote this algorithm by $\mathsf{Commit}^{\mathrm{RO}}_{\mathsf{M}}$.

Appendix B. Deploying ZK Web Applications

We provide a novel solution for running large-scale zero-knowledge proofs while maintaining space efficiency. Our compilation toolchain works as follows:

1) The user implements the NP-statement (or deterministic statement in the case of SNARKs) in a high-level language (C,C++,Python,Rust). It will be assumed that the program takes a witness w as input and the statement is hardcoded.[3]
2) The user uses an existing cross-compiler (Emscripten, Cargo, etc.) to compile the high-level program to WASM. The user can optionally run optimization passes on the generated WASM code (for example, using Binaryen) to reduce memory footprint.
3) The target Web Application embeds the WASM code of the Ligetron prover function for the WASM NP-statement. The application transmits the witness to the prover function and executes it to generate the proof.
4) The proof can be downloaded or transmitted across the Internet to a *verifying* application.
5) The verifying application can embed the Ligetron verifier WASM code along with the WASM-compiled NP statement. The Web Application can transmit them to the verifier function and execute it.

3. This is purely for simplicity. We could have the program input the public part, i.e the statement x, and the private part, i.e. the witness w.

Appendix C.
Meta-Review

C.1. Summary

The paper presents Ligetron, a proof system with very low prover overhead. Ligetron builds off Ligero, but through careful use of WASM for defining the proof statement on-the-fly and tightly coupling it to the steps to compute the proof, it avoids the memory overhead associated with proving a flattened circuit representation.

C.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Creates a New Tool to Enable Future Science

C.3. Reasons for Acceptance

1) The paper provides a valuable step forward in an established field. The approach for constructing a low overhead prover based on Ligero is clever and the underlying techniques are clearly explained. The performance results demonstrate a significant improvement in memory consumption over other state of the art solutions.
2) Creates a new tool to enable future science. The paper has implemented Ligetron which can serve as a basis for future research to further improve the system.