TrueSet: Faster Verifiable Set Computations*

Ahmed E. Kosba† Dimitrios Papadopoulos‡ Charalampos Papamanthou†
Mahmoud F. Sayed† Elaine Shi† Nikos Triandopoulos§

Abstract

Verifiable computation (VC) enables thin clients to efficiently verify the computational results produced by a powerful server. Although VC was initially considered to be mainly of theoretical interest, over the last two years, impressive progress has been made on implementing VC. Specifically, we now have open-source implementations of VC systems that can handle *all* classes of computations expressed either as circuits or in the RAM model. However, despite this very encouraging progress, new enhancements in the design and implementation of VC protocols are required in order to achieve truly practical VC for real-world applications.

In this work, we show that for functionalities that can be expressed efficiently in terms of set operations (e.g., a subset of SQL queries) VC can be enhanced to become drastically more practical: We present the design and prototype implementation of a novel VC scheme that achieves orders of magnitude speed-up in comparison with the state of the art. Specifically, we build and evaluate TrueSet, a system that can verifiably compute any polynomial-time function expressed as a circuit consisting of "set gates" such as *union*, *intersection*, *difference* and *set cardinality*. Moreover, TrueSet supports hybrid circuits consisting of both set gates and traditional arithmetic gates. Therefore, it does not lose any of the expressiveness of the previous schemes—this also allows the user to choose the most efficient way to represent different parts of a computation. By expressing set computations as polynomial operations and introducing a novel Quadratic Polynomial Program technique, TrueSet achieves prover performance speed-up ranging from **30x** to **150x** and yields up to **97**% evaluation key size reduction.

1 Introduction

Verifiable Computation (VC) is a cryptographic protocol that allows a client to outsource expensive computation tasks to a worker (e.g., a cloud server), such that the client can verify the result of the computation in less time than that required to perform the computation itself. Cryptographic approaches for VC [4, 5, 6, 11, 12, 13, 19] are attractive in that they require no special trusted hardware or software on the server, and can ensure security against arbitrarily malicious server behavior, including software/hardware bugs, misconfigurations, malicious insiders, and physical attacks.

Due to its various applications such as secure cloud computing, the research community has recently made impressive progress on Verifiable Computation, both on the theoretical and practical fronts. In particular, several recent works [2, 8, 22, 24, 25, 28] have implemented Verifiable

*This research was funded in part by NSF under grant numbers CNS-1314857, CNS-1012798 and CNS-1012910 and by a Google Faculty Research Award. The views and conclusions contained herein are those of the authors and should not be interpreted as representing funding agencies.

†University of Maryland, College Park. Email: akosba@cs.umd.edu, cpap@umd.edu, mfayoub@cs.umd.edu and elaine@cs.umd.edu.

‡Boston University. Email: dipapado@bu.edu.

§RSA Laboratories, Cambridge MA, USA and Boston University. Email: Nikolaos.Triandopoulos@rsa.com.

Computation for general computation tasks, and demonstrated promising evidence of its efficiency. Despite this encouraging progress, performance improvement of orders of magnitude is still required (especially on the time that the server takes to compute the proof) for cryptographic VC to become truly practical.

Existing systems for Verifiable Computation are built to accommodate any language in NP: Specifically, functions/programs are represented as either circuits (boolean or arithmetic) or sets of constraints and cryptographic operations are run on these representations. While such an approach allow us to express any polynomial-time computation, it is often not the most efficient way to represent common computation tasks encountered in practice. For example, Parno et al. [22] point out that the behavior of their construction deteriorates abruptly for functionalities that have "bad" arithmetic circuit representation and Braun et al. [8] recognize that the costs of their system are very high for the prover and the verifier's setup phase.

In order to reduce the practical cost of Verifiable Computation, in this paper we design and build TrueSet. TrueSet is an efficient and *provably secure* VC system that specializes in handling *set-centric* computation tasks. It allows us to model computation as a *set circuit*—a circuit consisting of a combination of set operators (such as intersection, union, difference and sum), instead of just arithmetic operations (such as addition and multiplication in a finite field). For computation tasks that can be naturally expressed in terms of set operations (e.g., a subset of SQL database queries), our experimental results suggest *orders-of-magnitude* performance improvement in comparison with existing VC systems such as Pinocchio [22]. We now present TrueSet's main contributions:

Expressiveness. TrueSet retains the expressiveness of existing VC systems, in that it can support arbitrary computation tasks. Fundamentally, since our set circuit can support intersection, union, and set difference gates, the set of logic is complete[1].

Additionally, in Section 4.4, we show that TrueSet can be extended to support circuits that have a mixture of arithmetic gates and set gates. We achieve this by introducing a "split gate" (which, on input a set, outputs the individual elements) and a "merge gate" (which has the opposite function of the split gate).

Input-specific running time. One important reason why TrueSet significantly outperforms existing VC systems in practice is that TrueSet achieves *input-specific* running time (during proof computation and key generation). Input-specific running time means that the running time of the prover is proportional to the size of the current input.

Achieving input-specific running time is not possible when set operations are expressed in terms of boolean or arithmetic circuits, where one must account for worst-case set sizes when building the circuit: For example, in the case of intersection, the worst case size of the output is the minimum size of the two sets; in the case of union, the worst case size of the output is the sum of their sizes. Note that this not only applies to the set that comprises the final outcome of the computation, but to every intermediate set generated during the computation. As a result, existing approaches based on boolean or arithmetic circuits incur a large blowup in terms of circuit size when used to express set operations. In this sense, TrueSet also achieves *asymptotic* performance gains for set-centric computation workloads in comparison with previous approaches.

TrueSet achieves input-specific running time by encoding a set of cardinality c as a polynomial of degree c (such encoding was also used in previous works, e.g., [17, 21]), and a set circuit as a circuit on polynomials, where every wire is a polynomial, and every gate performs polynomial addition or multiplication. As a result, per-gate computation time for the prover (including the

[1] Any function computable by boolean circuits can be computed by a set circuit: If one encodes the empty set as 0 and a fixed singleton set $\{s\}$ as 1, a union expresses the OR gate, an intersection expresses the AND gate and a set difference from $\{s\}$ expresses the NOT gate.

```
SELECT COUNT(UNIVERSITY.id)
    FROM UNIVERSITY JOIN CS ON UNIVERSITY.id = CS.id
```

Figure 1: An example of a `JOIN` *SQL query (between tables* **UNIVERSITY** *and* **CS***) that can be efficiently supported by* TrueSet*.* TrueSet *will implement* `JOIN` *with an intersection gate and* `COUNT` *with a cardinality gate.*

time for performing the actual computation and the time for producing the proof) is (quasi-)linear in the degree of the polynomial (i.e., cardinality of the actual set), and not proportional to the worst-case degree of the polynomial.

Finally, as in other VC systems, verifying in TrueSet requires work proportional to the size of inputs/outputs, but not in the running time of the computation.

Implementation and experimental results. We implemented TrueSet and documented its efficiency comparing it with a verifiable protocol that compiles a set circuit into an arithmetic circuit and then uses Pinocchio [22] on the produced circuit. In TrueSet the prover's running time is reduced by approximately **30x** for all set sizes of 64 elements or more. In particular, for a single intersection/union gate over 2 sets of 256 elements each, TrueSet improves the prover cost by nearly **150x**. We also show that, while other systems [22] cannot—in a reasonable amount of time—execute over larger inputs, TrueSet can scale to large sets, e.g., sets with cardinality of approximately 8000 (2^{13}), efficiently accommodating instances that are about **30x** larger than previous systems. Finally, TrueSet greatly reduces the evaluation key size, a reduction that can reach **97**% for some operations.

Applications. TrueSet is developed to serve various information retrieval applications that use set operations as a building block. For example, consider an SQL query that performs a `JOIN` over two tables and then computes `MAX` or `SUM` over the result of the join operation. TrueSet can model the join operation as an intersection and then use the split gate to perform the maximum or the summation/cardinality operation over the output of the join—see Figure 1. Other queries that TrueSet could model are advanced keyword search queries containing complicated filters that can be expressed as arbitrary combinations of set operations (union, intersection, difference) over an underlying data set. Finally, the computation of similarity measurements for datasets often employs set operations. One of the most popular measurements of this type, is the Jaccard index [16] which is computed for two sets, as the ratio of the *cardinalities* of their intersection and union, a computation that can be easily compiled with TrueSet.

Technical highlight. Our core technical construction is inspired by the recent *quadratic span and arithmetic programs* [13], which were used to implement VC for any boolean or arithmetic circuit. Since our internal representation is a polynomial circuit (as mentioned earlier), we invent *quadratic polynomial programs* (QPP). During the prover's computation, polynomials on the wires of the circuit are evaluated at a random point s—however, this takes place in the exponent of a bilinear group, in a way that the server does not learn s. Evaluating the polynomial at the point s in effect reduces the polynomial to a value—therefore one can now think of the polynomial circuit as a normal arithmetic circuit whose wires encode plain values. In this way, we can apply techniques resembling quadratic arithmetic programs. While the intuition may be summarized as above, designing the actual algebraic construction and formally proving its security is nonetheless challenging, and requires a non-trivial transformation of quadratic arithmetic programs.

1.1 Related Work

There has been a large amount of theoretical work on Verifiable Computation (VC): Micali [19] presented a scheme that can accommodate proofs for any language in NP. A more efficient approach is based on the construction of *succinct non-interactive arguments of knowledge* (SNARKs) [4, 5, 6, 13]. For the case of polynomial-time computable functions, protocols based on fully-homomorphic encryption (FHE) [11, 12] and attribute-based encryption (ABE) [23] have also been proposed. In general, the above schemes employ heavy cryptographic primitives and therefore are not very practical.

Recent works [2, 8, 22, 24, 25, 28] have made impressive progress toward implementations of some of the above schemes, showing practicality for particular functionalities. Unfortunately, the server's cost for proof computation remains too high to be considered for wide deployment in real-world applications.

The problem of verifying a circuit of set operations was first addressed in a recent work by Canetti et al. [9]. Their proofs are of size linear to the size of the circuit, without however requiring a preprocessing phase for each circuit. In comparison, our proofs are of constant size, once such a preprocessing step has been run.

Papamanthou et al. [21] presented a scheme that provides verifiability for a single set operation. However, one cannot accommodate more general set operations by repeatedly using their approach, since all intermediate set outputs are necessary for verification. This would lead to increased communication complexity.

A related scheme is also the scheme of Chung et al. [10]. This scheme uses Turing machines as the underlying computation model, the prover has inherently high complexity (e.g., linear in the size of the sets). Another work that combines verifiable computation with outsourcing of storage is [1] where a protocol for outsourced streaming datasets is proposed but the supported class of functionalities is restricted to arithmetic functions that can be expressed as polynomials of degree two.

2 Definitions

In this section we give some necessary definitions and introduce some terminology that is going to be useful in the rest of the paper.

Circuits of Sets and Polynomials. TrueSet uses the same computation abstraction as the one used in the VC scheme by Parno et al. [22]: A circuit. However, instead of field elements, the circuit wires now carry *sets*, and, instead of arithmetic multiplication and addition gates, our circuit has three types of gates: *Intersection*, *union* and *difference*. For the sake of presentation, the sets we are considering are simple sets, though our construction can be extended to support multisets as well. We therefore begin by defining a set circuit:

Definition 1 (Set circuit $\mathcal{C}$) *A set circuit $\mathcal{C}$ is a circuit that has gates that implement set union, set intersection or set difference over sets that have elements in a field $\mathbb{F}$.*

A set circuit is a tool that provides a clean abstraction of the computational steps necessary to perform a set operation. This structured representation will allow us to naturally encode a set operation into a number of execution conditions that are met when it is performed correctly. We stress that it is merely a theoretical abstraction and does not affect the way in which the computation is performed; the computing party can use its choice of efficient native libraries and architectures. In comparison, previous works that use arithmetic circuits to encode more general

computations, require the construction (or simulation) and evaluation of such a circuit, an approach that introduces an additional source of overhead.

As mentioned in the introduction, our main technique is based on mapping any set circuit $\mathcal{C}$ to a circuit $\mathcal{F}$ of polynomial operations, i.e., to a circuit that carries univariate polynomials on its wires and has polynomial multiplication and polynomial addition gates. We now define the polynomial circuit $\mathcal{F}$:

Definition 2 (Polynomial circuit $\mathcal{F}$) *A polynomial circuit $\mathcal{F}$ in a field $\mathbb{F}$ is a circuit that has gates that implement univariate polynomial addition and univariate polynomial multiplication over $\mathbb{F}$. We denote with d the number of multiplication gates of $\mathcal{F}$ and with N the number of input and output wires of $\mathcal{F}$. The inputs and output wires are indexed $1, \ldots, N$. The rest of the wires[2] are indexed $N+1, \ldots, m$.*

SNARKs. TrueSet's main building block is a primitive called *succinct non-interactive argument of knowledge* (SNARK) [13]. A SNARK allows a client to commit to a computation circuit C and then have a prover provide succinct cryptographic proofs that there exists an assignment on the wires w (which is called witness) such that the input-output pair $x = (\mathcal{I}, \mathcal{O})$ is valid.

As opposed to *verifiable computation* [23], a SNARK allows a prover to specify some wires of the input $\mathcal{I}$ as part of the witness w (this is useful when proving membership in an NP language, where the prover must prove witness existence). For this reason, SNARKs are more powerful than VC and therefore throughout the rest of the paper, we are going to show how to construct a SNARK for hierarchical set operations. In Appendix 6.6, we show how to use the SNARK construction to provide a VC construction as well and we also provide a scheme for VC over outsourced sets, where the server not only performs the computation, but also stores the sets for the client. We now give the SNARK definition, adjusted from [13].

Definition 3 (SNARK scheme) *A* SNARK *scheme consists of three PPT algorithms* (KeyGen, $\mathsf{Prove}, \mathsf{Verify}$) *defined as follows.*

1. $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^k, C)$. *The key generation algorithm takes as input the security parameter k and a computation circuit C; it outputs a public key pk, and a secret key sk.*

2. $\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$*: The prover algorithm takes as input the public key pk, an input-output pair $x = (\mathcal{I}, \mathcal{O})$, a valid witness w and it outputs a proof π.*

3. $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{sk}, x, \pi)$*: Given the key sk, a statement x and a proof π, the verification algorithm outputs 0 or 1.*

We say that a SNARK is publicly-verifiable if $\mathsf{sk} = \mathsf{pk}$. In this case, proofs can be verified by anyone with pk. Otherwise, we call it a secretly-verifiable SNARK, in which case only the party with sk can verify.

There are various properties that a SNARK should satisfy. The most important one is *soundness*. Namely, no PPT adversary should be able to output a verifying proof π for an input-output pair $x = (\mathcal{I}, \mathcal{O})$ that is not consistent with C. All the other properties of SNARKs are described formally in Appendix 6.3.

[2] These wires include free wires (which are inputs only to multiplication gates) and the outputs of the internal multiplication gates (whose outputs are not outputs of the circuit). The set of these wires is denoted with I_m and has size at most $3d$.

3 A SNARK for Polynomial Circuits

In their recent seminal work, Gennaro et al. [13] showed how to compactly encode computations as quadratic programs, in order to derive very efficient SNARKs. Specifically, they show how to convert any arithmetic circuit into a comparably-sized Quadratic Arithmetic Program (QAP), and any Boolean circuit into a comparably-sized Quadratic Span Program (QSP).

In this section we describe our SNARK construction for polynomial circuits. The construction is a modification of the optimized construction for arithmetic circuits that was presented by Parno et al. [22] (Protocol 2) and which is based on the original work of Gennaro et al. [13]. Our extension accounts for univariate polynomials on the wires, instead of just arithmetic values. We therefore need to define a *quadratic polynomial program*:

Definition 4 (Quadratic Polynomial Program (QPP)) *A QPP $\mathcal{Q}$ for a polynomial circuit $\mathcal{F}$ contains three sets of polynomials $\mathcal{V} = \{v_k(x)\}, \mathcal{W} = \{w_k(x)\}, \mathcal{Y} = \{y_k(x)\}$ for $k = 1, \ldots, m$ and a target polynomial $\tau(x)$. We say that $\mathcal{Q}$ computes $\mathcal{F}$ if: $c_1(z), c_2(z), \ldots, c_N(z)$ is a valid assignment of $\mathcal{F}$'s inputs and outputs iff there exist polynomials $c_{N+1}(z), \ldots, c_m(z)$ such that $\tau(x)$ divides $p(x, z)$ where*

$$p(x,z) \quad = \quad \left(\sum_{k=1}^{m} c_k(z) v_k(x)\right)\left(\sum_{k=1}^{m} c_k(z) w_k(x)\right) - \left(\sum_{k=1}^{m} c_k(z) y_k(x)\right) .$$

We also define the degree of $\mathcal{Q}$ to equal the degree of $\tau(x)$.

The main difference of the above quadratic program with the one that was presented in [22] is the fact that we introduce another variable z in the polynomial $p(x, z)$ representing the program (hence we need to account for bivariate polynomials, instead of univariate), which is going to account for the polynomials on the wires of the circuit.

We now show how to construct a QPP $\mathcal{Q}$ for a polynomial circuit. The polynomials in $\mathcal{V}, \mathcal{W}, \mathcal{Y}$ and the polynomial $\tau(x)$ are computed as follows. Let $r_1, r_2, \ldots, r_d$ be random elements in $\mathbb{F}$. First, set $\tau(x) = (x - r_1)(x - r_2)\ldots(x - r_d)$ and compute the polynomial $v_k(x)$ such that $v_k(r_i) = 1$ iff wire k is the *left input* of multiplication gate i, otherwise $v_k(r_i) = 0$. Similarly, $w_k(r_i) = 1$ iff wire k is the *right* input of multiplication gate i, otherwise $w_k(r_i) = 0$ and $y_k(r_i) = 1$ iff wire k is the *output* of multiplication gate i, otherwise $y_k(r_i) = 0$. For example, consider the circuit of Figure 2 that has five inputs and one output and its wires are numbered as shown in the figure (gates take the index of the their output wire). Then $\tau(x) = (x - r_6)(x - r_7)$. For v_k we require that $v_k(r_6) = 0$ except for $v_2(r_6) = 1$, since the second wire is the only left input for the sixth gate, and $v_k(r_7) = 0$ except for $v_1(r_7)$ and $v_6(r_7)$ which are 1, since the first and sixth wire contribute as left inputs to gate 7. Right input polynomials w_k are computed similarly and output polynomials y_k are computed such that $y_6(r_6) = y_7(r_7) = 1$; all other cases are set to 0.

To see why the above QPP computes $\mathcal{F}$, let us focus on a single multiplication gate g, with k_1 being its output wire and k_2 and k_3 be its left and right input wires respectively. Due to the divisibility requirement, it holds $p(r_i, z) = 0$ for $i = 1, \ldots, d$, hence Equation 3.1 will give $(\sum_{k=1}^{m} c_k(z)v_k(r_g))(\sum_{k=1}^{m} c_k(z)w_k(r_g)) = (\sum_{k=1}^{m} c_k(z)y_k(r_g))$. Now, from the way the polynomials v_k, w_k, y_k were defined above, most terms are 0 and what remains is $c_{k_2}(z)v_{k_2}(r_g) \cdot c_{k_3}(z)w_{k_3}(r_g) = c_{k_1}(z)y_{k_1}(r_g)$ or else $c_{k_2}(z) \cdot c_{k_3}(z) = c_{k_1}(z)$, which is the definition of a multiplication gate. More formally (the proof is in the appendix):

Lemma 1 *The above QPP $\mathcal{Q}$ computes $\mathcal{F}$.*

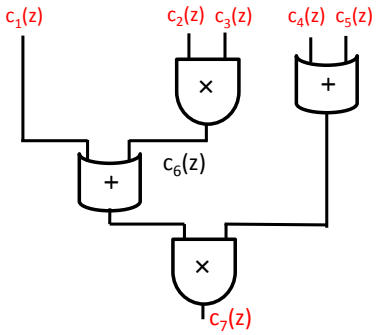


Figure 2: A sample polynomial circuit.

We now give an efficient SNARK construction for polynomial circuits based on the above QPP. Recall that a polynomial circuit $\mathcal{F}$ has d multiplication gates and m wires, the wires $1, \ldots, N$ occupy inputs and outputs and set $I_m = \{N+1, \ldots, m\}$ represents the internal wires, where $|I_m| \leq 3d$. Also, we will denote with n_i the degree of polynomial on wire i. We also set n be an upper bound on the degrees of the polynomials on $\mathcal{F}$'s wires.

3.1 Intuition of Construction

The SNARK construction that we present works as follows. First, the key generation algorithm KeyGen produces a "commitment" to the polynomial circuit $\mathcal{F}$ by outputting elements that relate to the internal set of wires I_m of the QPP $\mathcal{Q} = (\mathcal{V}, \mathcal{W}, \mathcal{Y}, t(x))$ as the public key. These elements encode bivariate polynomials in the exponent, evaluated at randomly chosen points t and s, to accommodate for the fact that circuit $\mathcal{F}$ encodes operations over univariate polynomials and not just arithmetic values (as is the case with [13]).

According to what we described in the previous section, for the prover to prove that an assignment $c_1(z), c_2(z), \ldots, c_N(z)$ of polynomials on input/output wires is valid, it suffices to prove there exist polynomials $c_{N+1}(z), \ldots, c_m(z)$ corresponding to assignments on the internal wires, such that the polynomial $p(x, z)$ from Relation 3.1 have roots $r_1, r_2, \ldots, r_d$. In other words, the following should hold for some polynomial $h(x, z)$:

$$p(x, z) = h(x, z)\tau(x)\,. \tag{3.1}$$

In order to prove the above, the prover first "solves" the circuit and computes the polynomials $c_1(z), c_2(z), \ldots, c_m(z)$ that correspond to the correct assignments on the wires. Then he uses these polynomials and the public evaluation key (i.e., the circuit "commitment") to compute the following three types of terms (which comprise the actual proof). The detailed computation of these values is described in Section 3.2.

- **Extractability terms.** These terms declare three polynomials in the exponent, namely $\sum_{k=N+1}^{m} c_k(z)v_k(x)$, $\sum_{k=N+1}^{m} c_k(z)w_k(x)$, and $\sum_{k=N+1}^{m} c_k(z)y_k(x)$. Specifically, these polynomials correspond to the internal wires since the verifier can fill in the parts for the input and output wires.

  The above terms are engineered to allow extractability using a knowledge assumption. In particular, given these terms, there exists a polynomial-time extractor that can, with overwhelming probability, recover the assignment $c_{N+1}(z), \ldots, c_m(z)$ on internal wires. This proves the existence of $c_{N+1}(z), \ldots, c_m(z)$.

- **Consistency check terms.** Extraction is done separately for terms related to polynomials $\sum_{k=N+1}^{m} c_k(z)v_k(x)$, $\sum_{k=N+1}^{m} c_k(z)w_k(x)$, and $\sum_{k=N+1}^{m} c_k(z)y_k(x)$. We therefore require a set of consistency check terms to ensure that the extracted $c_{N+1}(z), \ldots, c_m(z)$ polynomials are consistent for the above $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{Y}$ terms—otherwise, the same wire can have ambiguous assignments.

- **Divisibility check term.** Finally, the divisibility check term is to ensure that the above divisibility check corresponding to Equation 3.1, holds for the polynomial

$$\left(\sum_{k=1}^{m} c_k(z)v_k(x)\right)\left(\sum_{k=1}^{m} c_k(z)w_k(x)\right) - \left(\sum_{k=1}^{m} c_k(z)y_k(x)\right)$$

  declared earlier by the extractability terms.

3.2 Concrete Construction

We now give the algorithms of our SNARK construction, following the definition of SNARKs (see Definition 3). In comparison with the QSP and QAP constructions [13, 22], one difficulty arises in our setting when working with polynomials on wires. In essence, to express a polynomial $c_k(z)$ on a wire in our construction, we evaluate the polynomial at a committed point $z = t$. In existing QSP and QAP constructions [13, 22], the prover knows the cleartext value on each wire when constructing the proof. However, in our setting, the prover does not know what t is, and hence cannot directly evaluate the polynomials $c_k(z)$'s on each wire. In fact, security would be broken if the prover knows the value of the polynomials at $z = t$.

To overcome this problem, we have to include more elements in the evaluation key which contains the exponent powers of the variable t (see the evaluation key below). In this way, the prover will be able to evaluate $c_k(t)$ in the exponent, without ever learning the value t. We now give the algorithms:

$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\mathcal{F}, 1^k)$: Let $\mathcal{F}$ be a polynomial circuit. Build the corresponding QPP $\mathcal{Q} = (\mathcal{V}, \mathcal{W}, \mathcal{Y}, t(x))$ as above. Let e be a non-trivial bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$, and let g be a generator of $\mathbb{G}$. $\mathbb{G}$ and $\mathbb{G}_T$ have prime order p. Pick $s, t, r_v, r_w, \alpha_v, \alpha_w, \alpha_y, \beta, \gamma$ from $\mathbb{Z}_p$ and set $r_y = r_v r_w$ and $g_v = g^{r_v}$, $g_w = g^{r_w}$ and $g_y = g^{r_y}$. The public evaluation key $\mathsf{EK}_{\mathcal{F}}$ is

1. $\{g_v^{t^i v_k(s)}, g_w^{t^i w_k(s)}, g_y^{t^i y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
2. $\{g_v^{t^i \alpha_v v_k(s)}, g_w^{t^i \alpha_w w_k(s)}, g_y^{t^i \alpha_y y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
3. $\{g_v^{t^i \beta\cdot v_k(s)} g_w^{t^i \beta\cdot w_k(s)} g_y^{t^i \beta\cdot y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
4. $\{g^{t^i s^j}\}_{(i,j)\in[2n]\times[d]}$.

The verification key $\mathsf{VK}_{\mathcal{F}}$ consists of the values

$$g, g^{\alpha_v}, g^{\alpha_w}, g^{\alpha_y}, g^{\gamma}, g^{\beta\gamma} g_y^{t(s)}$$

and the set $\{g_v^{t^i v_k(s)}, g_w^{t^i w_k(s)}, g_y^{t^i y_k(s)}\}_{(i,k)\in[n]\times[N]}$. Note $\mathsf{VK}_{\mathcal{F}}$ and $\mathsf{EK}_{\mathcal{F}}$ are the public key pk of the SNARK. Our SNARK is publicly verifiable, hence $\mathsf{sk} = \mathsf{pk}$.

$\pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w)$: The input x contains input polynomials u and output polynomials y and the witness w (which contains assignments of polynomials on the internal wires). Let $c_k(z)$ be the polynomials on the circuit's wires such that $y = \mathcal{F}(u, w)$. Let $h(x, z)$ be the polynomial such that $p(x, z) = h(x, z) \cdot \tau(x)$. The proof is computed as follows:

1. *(Extractability terms)* $g_v^{v_m(s,t)}$, $g_w^{w_m(s,t)}$, $g_y^{y_m(s,t)}$, $g_v^{\alpha_v v_m(s,t)}$, $g_w^{\alpha_w w_m(s,t)}$, $g_y^{\alpha_y y_m(s,t)}$.
2. *(Consist. check term)* $g_v^{\beta \cdot v_m(s,t)} g_w^{\beta \cdot w_m(s,t)} g_y^{\beta \cdot y_m(s,t)}$.
3. *(Divisibility check term)* $g^{h(s,t)}$,

where $v_m(x,z) = \sum_{k \in I_m} c_k(z) v_k(x)$, $w_m(x,z) = \sum_{k \in I_m} c_k(z) w_k(x)$ and $y_m(x,z) = \sum_{k \in I_m} c_k(z) y_k(x)$. Note that the term $g_v^{\beta \cdot v_m(s,t)} g_w^{\beta \cdot w_m(s,t)} g_y^{\beta \cdot y_m(s,t)}$ can be computed from the terms

$$\{g_v^{t^i \beta \cdot v_k(s)} g_w^{t^i \beta \cdot v_k(s)} g_y^{t^i \beta \cdot y_k(s)}\}_{(i,k) \in [n] \times I_m}$$

of the public key pk.

$\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{pk}, x, \pi)$: Parse the proof π as

1. $\gamma_v, \gamma_w, \gamma_y, \kappa_v, \kappa_w, \kappa_y$.
2. Λ.
3. γ_h.

First, verify all three α terms: $e(\gamma_v, g^{\alpha_v}) \stackrel{?}{=} e(\kappa_v, g) \wedge e(\gamma_w, g^{\alpha_w}) \stackrel{?}{=} e(\kappa_w, g) \wedge e(\gamma_y, g^{\alpha_y}) \stackrel{?}{=} e(\kappa_y, g)$. Then verify the divisibility requirement:

$$e(\lambda_v \cdot \gamma_v, \lambda_w \cdot \gamma_w) / e(\lambda_y \cdot \gamma_y, g) \stackrel{?}{=} e(\gamma_h, g^{\tau(s)}),$$

where $\lambda_v = g^{\sum_{k \in [N]} c_k(t) v_k(s)}$, $\lambda_w = g^{\sum_{k \in [N]} c_k(t) w_k(s)}$, $\lambda_y = g^{\sum_{k \in [N]} c_k(t) y_k(s)}$. Finally verify the β term:

$$e(\gamma_v \cdot \gamma_w \cdot \gamma_y, g^{\beta\gamma}) \stackrel{?}{=} e(\Lambda, g^{\gamma}).$$

3.3 Asymptotic Complexity and Security

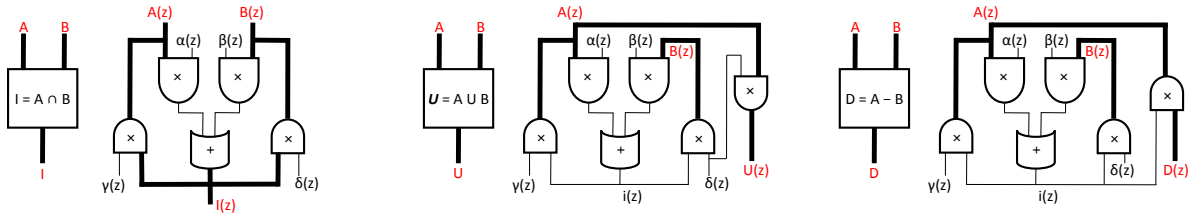
In this section we analyze the asymptotic complexity of our SNARK construction for polynomial circuits. We also state the security of our scheme.

KeyGen: It is easy to see that the computation time of KeyGen is $O(n|I_m| + nd + nN) = O(dn)$.

Prove: Let T be the time required to compute the polynomials $c_i(z)$ for $i = 1, \ldots, m$ and let n_i be the degree of the polynomial $c_i(z)$ for $i = 1, \ldots, m$. The computation of each $g^{c_i(z) v_i(x)}$ (similarly for $g^{c_i(z) w_i(x)}$ and $g^{c_i(z) y_i(x)}$) for $i \in I_m$ takes $O(n_i)$ time (specifically, $7 \cdot \sum n_i$ exponentiations are required to compute all the proof), since one operation per coefficient of $c_i(z)$ is required. Then multiplication of $|I_m|$ terms is required. Therefore the total time required is

$$O\left(T + \sum_{i \in I_m} n_i + |I_m|\right) = O\left(T + d\nu\right),$$

where $\nu = \max_{i=1,\ldots,m}\{n_i\}$ is the maximum degree of the polynomials over the wires and since $|I_m| \leq 3d$. To compute $p(x,z)$, first the degree d polynomials $v_i(x), w_i(x), y_i(x)$ for $i = 1, ..., m$ are parsed in time $O(dm)$. Then $p(x,z)$ is computed according to Equation 1; each summation term is computed in time $O(d\nu)$ with naive bivariate polynomial multiplication and then they are summed for total complexity of $O(md\nu)$. For the division, note that $p(x,z)$ has maximum degree in z equal to 2ν and maximum degree in x equal to $2d$. To do the division, we apply "the change of variable trick". We set $z = x^{2 \times (2d)+1}$ and therefore turn $p(x,z)$ into a polynomial of one variable x, namely the polynomial $p(x, x^{2 \times (2d)+1})$. Therefore the dividend now has maximum degree $2\nu(4d+1) + 2d$



(a) (b) (c)

Figure 3: Set circuits for intersection (a), union (b) and difference (c) expressed as polynomial circuits with loops using Lemma 2, Corollary 1 and Corollary 2.

while the divisor has still degree d. By using FFT, we can do such division in $O(d\nu \log(d\nu))$ time. Therefore the total time for Prove is $O\left(T + d\nu \log(d\nu) + md\nu\right)$.

Verify: The computation of each element $g^{c_i(z)v_i(x)}$ (similarly for $g^{c_i(z)w_i(x)}$ and $g^{c_i(z)y_i(x)}$) for $i = 1, \ldots, N$ takes $O(n_i)$ time, since one operation per coefficient of $c_i(z)$ is required. Then multiplication of N terms is required. Therefore the total time required is $O(\sum_{i\in[N]} n_i)$, proportional to the size of the input and output.

We now have the following result. Its proof of security can be found in Appendix 6.5 and the involved assumptions in Appendix 6.2.

Theorem 1 (Security of the SNARK for $\mathcal{F}$) *Let $\mathcal{F}$ be a polynomial circuit with d multiplication gates. Let n be an upper bound on the degrees of the polynomials on the wires of $\mathcal{F}$ and let $q = 4d + 4$. The construction above is a SNARK under the $2(n+1)q$-PKE, the $(n+1)q$-PDH and the $2(n+1)q$-SDH assumptions.*

4 Efficient SNARKs for Set Circuits

In this section, we show how to use the SNARK construction for polynomial circuits that was presented in the previous section to build a SNARK for set circuits.

To achieve that, we do the following: We first define a mapping from sets to polynomials (see Definition 5—such representation was also used in prior work, e.g., the work of Kissner and Song [17]). Then we express the correctness of the operations between two sets (e.g., set intersection) as constraints between the polynomials produced from this mapping (e.g., see Lemma 2). For a set operation to be correct, these constraints must be satisfied simultaneously. To capture that, we represent all these constraints with a circuit with loops, where a wire can participate in more than one constraints (see Figure 3).

4.1 Expressing Sets with Polynomials

We first show how to represent sets and set operations with polynomials and polynomial operations. This representation is key to achieving *input-specific time*, since we can represent a set with a polynomial (evaluated at a random committed point), regardless of the cardinality of the set.

Given a set, we first define the *characteristic polynomial* of a set.

Definition 5 (Characteristic polynomial) *Let A be a set of elements $\{a_1, a_2, \ldots, a_n\}$ in $\mathbb{F}$. We define its characteristic polynomial as $\mathsf{A}(z) = (z + a_1)(z + a_2)\ldots(z + a_n)$.*

We now show the relations between set operations and polynomial operations. Note that similar relations were used by Papamanthou et al. [21] in prior work.

Lemma 2 (Intersection constraints) *Let* A, B *and* I *be three sets of elements in* $\mathbb{F}$. *Then* $\mathsf{I} = \mathsf{A} \cap \mathsf{B}$ *iff there exist polynomials* $\alpha(z)$, $\beta(z)$, $\gamma(z)$ *and* $\delta(z)$ *such that*

1. $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = \mathsf{I}(z)$.
2. $\gamma(z)\mathsf{I}(z) = \mathsf{A}(z)$.
3. $\delta(z)\mathsf{I}(z) = \mathsf{B}(z)$.

Proof: ($\Rightarrow$) If $\mathsf{I} = \mathsf{A} \cap \mathsf{B}$, it follows that (i) the great common divisor of polynomials $\mathsf{A}(z)$ and $\mathsf{B}(z)$ is $\mathsf{I}(z)$, therefore, by Bézout's identity, there exist polynomials $\alpha(z)$ and $\beta(z)$ such that (i) $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = \mathsf{I}(z)$; (ii) $\mathsf{I}(z)$ divides $\mathsf{A}(z)$ and $\mathsf{B}(z)$, therefore there exist polynomials $\gamma(z)$ and $\delta(z)$ such that $\gamma(z)\mathsf{I}(z) = \mathsf{A}(z)$ and $\delta(z)\mathsf{I}(z) = \mathsf{B}(z)$.

($\Leftarrow$) Let A, B and I be sets. Suppose there exist polynomials $\alpha(z)$, $\beta(z)$, $\gamma(z)$ and $\delta(z)$ such that (1), (2) and (3) are true. By replacing (2) and (3) into (1), we get that $\alpha(z)$ and $\beta(z)$ do not have any common factor, therefore $\mathsf{I}(z)$ is the greatest common divisor of $\mathsf{A}(z)$ and $\mathsf{B}(z)$ and therefore $\mathsf{A} \cap \mathsf{B} = \mathsf{I}$. ■

Corollary 1 (Union constraints) *Let* A, B *and* U *be three sets of elements in* $\mathbb{F}$. *Then* $\mathsf{U} = \mathsf{A} \cup \mathsf{B}$ *iff there exist polynomials* $i(z)$, $\alpha(z)$, $\beta(z)$, $\gamma(z)$ *and* $\delta(z)$ *such that*

1. $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = i(z)$.
2. $\gamma(z)i(z) = \mathsf{A}(z)$.
3. $\delta(z)i(z) = \mathsf{B}(z)$.
4. $\delta(z)\mathsf{A}(z) = \mathsf{U}(z)$.

Corollary 2 (Difference constraints) *Let* A, B *and* D *be three sets of elements in* $\mathbb{F}$. *Then* $\mathsf{D} = \mathsf{A} - \mathsf{B}$ *iff there exist polynomials* $i(z)$, $\alpha(z)$, $\beta(z)$, $\gamma(z)$ *and* $\delta(z)$ *such that*

1. $\alpha(z)\mathsf{A}(z) + \beta(z)\mathsf{B}(z) = i(z)$.
2. $\mathsf{D}(z)i(z) = \mathsf{A}(z)$.
3. $\delta(z)i(z) = \mathsf{B}(z)$.

4.2 Compiling Set Circuits into Polynomial Circuits

Polynomial circuits with loops. To compile a set circuit into a circuit on polynomials, we need to check that the constraints in Lemma 2 and Corollaries 1 and 2 simultaneously satisfy for all intersection, union, and set difference gates respectively. Doing this in a straightforward manner seems to require implementing a boolean AND gate using polynomial algebra, which introduces an unnecessary representation overhead.

We use a simple idea to avoid this issue, by introducing polynomial circuits with *loops*. This means that the circuit's wires, following the direction of evaluation, can contain loops, as shown in Figure 3. When a circuit contains loops, we require that there exist an assignment for the wires such that every gate's inputs and output are consistent. It is not hard to see that we can build a QPP for a polynomial circuit with loops.

From set circuits to polynomial circuits. Suppose we have a set circuit $\mathcal{C}$, as defined in Definition 1. We can compile circuit $\mathcal{C}$ into a polynomial circuit with loops $\mathcal{F}$ as follows:

1. Replace every intersection gate g_I with the circuit of Figure 3(a), which implements the constraints in Lemma 2. Note that 6 additional wires per intersection gate are introduced during this compilation, 4 of which are free wires. Also, for each intersection gate, 4 polynomial multiplication gates are added.

2. Replace every union gate g_U of $\mathcal{C}$ with the circuit of Figure 3(b), which implements the set of constraints in Corollary 1. Note that 7 additional wires per union gate are introduced during this compilation, 3 of which are free wires. Also, for each union gate, 5 polynomial multiplication gates are added.

3. Replace every difference gate g_D of $\mathcal{C}$ with the circuit of Figure 3(c), which implements the set of constraints in Corollary 2. Note that 7 additional wires per union gate are introduced during this compilation, 3 of which are free wires. Also, for each difference gate, 5 polynomial multiplication gates are added.

4.3 Asymptotic Complexity and Security

Let $\mathcal{C}$ be the circuit for set operations that has d gates (out of which d_1 are intersection gates and d_2 are union and difference gates) and N inputs and outputs. After compiling $\mathcal{C}$ into an polynomial circuit with loops $\mathcal{F}$, we end up with a circuit $\mathcal{F}$ has $4d_1 + 5d_2$ multiplication gates since each intersection gate introduces 4 multiplication gates and each union or difference gate introduce 5 multiplication gates.

Therefore, a SNARK for set circuits with $d = d_1 + d_2$ gates can be derived from a SNARK for polynomial circuits having $4d_1 + 5d_2$ multiplication gates. Note that the complexity of algorithm Prove for the SNARK for set circuits is $O(d\nu \log^2 \nu \log\log \nu)$ because the prover runs the extended Euclidean algorithm to compute the polynomials on the free wires, which takes $O(t \log^2 t \log\log t)$ time, for t-degree polynomials as inputs.

Theorem 2 (Security of the SNARK for $\mathcal{C}$) *Let $\mathcal{C}$ be a set circuit that has d total gates and N total inputs and outputs. Let n be an upper bound on the cardinalities of the sets on the wires of $\mathcal{C}$ and let $q = 16d_1 + 20d_2 + 4$, where d_1 is the number of intersection gates and d_2 is the number of union and difference gates ($d = d_1 + d_2$). The construction above is a SNARK construction for the set circuit $\mathcal{C}$ under the $2(n+1)q$-PKE, the $(n+1)q$-PDH and the $2(n+1)q$-SDH assumptions.*

We note here that there do exist known SNARK constructions for languages in NP that have excellent asymptotic behavior and are *input-specific*, e.g., the work of Bitansky et al. [5], based on recursive proof composition. Therefore, in theory, our SNARK asymptotics are the same with the ones by Bitansky et al. [5] (when applied to the case of set operations).

However, the concrete overhead of such techniques remains high; in fact, for most functionalities it is hard to deduce the involved constants. In comparison, with our approach, we can always deduce an upper bound on the number of necessary operations involved. We give a tight complexity analysis of our approach in Appendix 6.7.

4.4 Handling More Expressive Circuits

As discussed in the introduction, by moving from QAPs to QPPs our scheme is not losing anything in expressiveness. However, in order to be efficient for set operations, so far we explicitly discussed set circuits that only consist of set gates. Ideally, we want to be able to efficiently accommodate "hybrid" circuits that consist both of set and arithmetic operations in an optimally tailored approach.

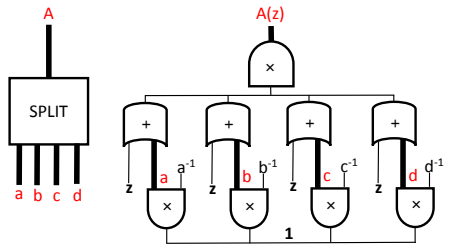


Figure 4: Implementation of a split gate for the set $\mathsf{A} = \{a, b, c, d\}$. *The elements* z *and* 1 *on the wires are hard-coded in the circuit during setup. All other polynomials on the wires are computed by the prover.*

In this section we show how, by constructing a *split* gate (and a *merge* gate) that upon input a set A outputs its elements a_i, we gain some "backwards compatibility" with respect to QAPs. In particular, this allows us to compute on the set elements themselves, e.g., performing `MAX` or `COUNT`. Also, using techniques described in [22], one can go one step below in the representation hierarchy and represent a_i's in binary form which yields, for example, more efficient comparison operations.

Hence we produce here a truly complete toolkit that a delegating client can use for an elaborate computation, in a way that allows him both to be more efficient for the part corresponding to set operations and at the same time perform arithmetic and bit operations in an optimal way, choosing different levels of abstraction for different parts of the circuit.

Zero-degree assertion gate. Arithmetic values can be naturally interpreted as zero-degree polynomials. Since we want to securely accommodate both polynomials and arithmetic values in our circuit, we need to construct a gate that will constrain the values of some wires to arithmetic values. For example, we need to assure that the outputs of a split gate are indeed numbers (and not higher degree polynomials). The following lemma is going to be useful for that.

Lemma 3 (Zero-degree constraints) *Let* $p(z)$ *be a univariate polynomial in* $\mathbb{F}[z]$. *The degree of* $p(z)$ *is 0 iff* $\exists$ *polynomial* $q(z)$ *in* $\mathbb{F}[z]$ *such that* $p(z)q(z) = 1$.

Proof: ($\Rightarrow$) Every zero-degree polynomial $q(z) \in \mathbb{F}[z]$ also belongs in $\mathbb{F}$. Since every element in $\mathbb{F}$ has an inverse, the claim follows. ($\Leftarrow$) Assume now that $p(z)q(z) = 1$. Since polynomial 1 is of degree 0, $p(z)q(z)$ must also be of degree 0. By polynomial multiplication, we know that $p(z)q(z)$ has degree $deg(p(z)) + deg(q(z))$. Therefore this can only hold if $deg(p(z)) = deg(q(z)) = 0$. ■

This simple gate consists of a multiplication gate between polynomial $p(z)$ and an auxiliary input $q(z)$ computed by the server and the output is set to the (hard-coded) polynomial 1. If the input is indeed a zero-degree polynomial, by the above Lemma, $q(z)$ is easily computable by the server (an inverse computation in $\mathbb{F}$).

Split gate. A split gate, depicted in Figure 4, operates as follows. On input a wire with value $\mathsf{A}(z)$, it outputs n wires with the individual elements a_i. First, each of the wires carrying a_i is connected to a degree-zero assertion gate. This will make sure that these wires carry arithmetic values. Second, each of these wires is used as an input to an addition gate, with the other input being the degree-one polynomial z. Then the outputs of all the addition gates are multiplied together and the output of the multiplication is connected to the wire carrying $\mathsf{A}(z)$.

Split gate with variable number of outputs. In the above we assumed that the split gate can have a fixed number of outputs, n. However, the number of outputs can vary. To accommodate this, we assume that n is an upper bound on the number of outputs of a split gate. Now, for each of the n output wires, we introduce an indicator variable ν_i (picked by the prover) such that if $\nu_i = 1$, this output wire is occupied and carries an arithmetic value, otherwise $\nu_i = 0$. Then, in the

split gate of Figure 4, instead of computing $\prod_{i=1}^{n}(z+a_i)$ we compute

$$\prod_{i=1}^{n}[\nu_i(z+a_i)+(1-\nu_i)]\,.$$

Note here that an additional restriction we need to impose is that $\nu_i \in \{0,1\}$. Fortunately this can be checked very easily by adding one self-multiplication gate and a loop wire for each value that enforces the condition $\nu_i \cdot \nu_i = \nu_i$ that clearly holds iff $\nu_i = 0$ or 1.

Cardinality gate. One immediate side-effect of our construction for split gates with variable number of outputs, is that it indicates a way to construct another very important type of gate, namely *cardinality* gate. Imagine for example a computation where the requested output is not a set but only its cardinality (e.g., a `COUNT` SQL-query). A cardinality gate is implemented exactly like a split gate, however it only has a single output wire that is computed as $\sum_i \nu_i$, using $n-1$ addition gates over the ν_i wires.

Merge gate. Finally, the *merge* gate upon input n wires carrying numerical values a_i, outputs a single wire that carries them as a set (i.e., its characteristic polynomial).

The construction of this gate is conceptually very similar to that of the split gate, only in reverse order. First the input wires are tested to verify that they are of degree 0 with n zero-degree assertion gates. Following that, these wires are used as input for union gates, taken in pairs, in an iterative manner (imagine a binary tree of unions with n leaves and the output set at the root). It should be noted that for computations that cater for multi-sets, this last step can be replaced by multiplication gates that provide a more efficient solution.

5 Evaluation

We now present the evaluation of TrueSet comparing its performance with Pinocchio [22], which is the state-of-the-art general VC scheme (already reducing computation time by orders-of-magnitude when compared with previous implementations). We also considered alternative candidates for comparison such as Pantry [8] which is specialized for stateful computations. Pantry is theoretically more efficient than Pinocchio, as it can support a RAM-based $O(n \log n)$-time algorithm for computing set intersection or $O(n)$-time algorithm when the input sets are sorted, instead of the circuit-based $O(n \log^2 n)$ or $O(n^2)$ algorithms that Pinocchio supports. However, evaluation showed that Pantry requires considerable proof construction time, due to the expensive memory-based operations (e.g., 30 seconds for a single verifiable *put* operation in a memory of 8192 addresses) which results in a large constant increasing the proof time significantly in the scale of our experiments. Therefore, we chose to compare only with Pinocchio, although Pantry *asymptotically* supports more efficient verifiable programs in comparison with Pinocchio.

In our experiments, we analyze the performance of TrueSet both for the case of a single set operation and multiple set operations. We begin by presenting the details of our implementation and the evaluation environment and then we present the performance results.

5.1 Implementation

We built TrueSet by extending Pinocchio's C++ implementation so that it can handle set circuits, with the special set gates that we propose. However, since the original implementation of Pinocchio used efficient libraries for pairing-based cryptography and field manipulation that are not available for public use (internal to Microsoft), the first step was to replace those libraries with available

free libraries that have similar characteristics. In particular, we used the Number Theory Library (NTL) [26] along with the GNU Multi-Precision (GMP) library [14] for polynomial arithmetic, in addition to an efficient free library for ate-pairing over Barreto-Naehrig curves [3], in which the underlying BN curve is $y^2 = x^3 + 2$ over a 254-bit prime field $\mathbb{F}_p$ that maintains a 126 bit-level of security. As in Pinocchio, the size of the cryptographic proof produced by our implementation is typically equal to 288 bytes in all experiments regardless of the input or circuit sizes.

TrueSet's executable receives an input file describing a set circuit that contains one or more of the set gates described earlier. The executable compiles the circuit to a QPP in two stages. In the first stage, the set gates are transformed into their equivalent representation using polynomial multiplication and addition gates, as in Figures 3 and 4, and then the QPP is formed directly in the second stage by generating the roots, and calculating the V, W and Y polynomials.

Optimizations. For a fair comparison, we employ the same optimizations used for reducing the exponentiation overhead in Pinocchio's implementation. Concerning polynomial arithmetic, Pinocchio's implementation uses an FFT approach to reduce the polynomial multiplication costs. In our implementation, we use the NTL library, which already provides an efficient solution for polynomial arithmetic based on FFT [27].

In addition to the above, the following optimizations were found to be very useful when the number of set gates is high, or when the set split gate is being used.

1) For key generation, we reduce the generated key size by considering the maximum polynomial degree that can appear on each wire, instead of assuming a global upper bound on the polynomial degree for all wires (as described in previous sections). This can be calculated by assuming a maximum cardinality of the sets on the input wires, and then iterating over the circuit wires to set the maximum degree per wire in the worst case, e.g. the sum of the worst case cardinalities of the input sets for the output of a union gate, and the smaller for intersections.

2) The NTL library does not provide direct support for bivariate polynomial operations, needed to calculate $h(x, z)$ through division of $p(x, z)$ by $\tau(x)$. Hence, instead of doing a naive $O(n^2)$ polynomial division, we apply the change-of-variable trick discussed in Section 3.3 to transform bivariate polynomials into univariate ones that can be handled efficiently with NTL FFT operations.

3) Finally, calculation of the coefficients of the characteristic polynomial corresponding to the output is done by the prover and not by the verifier. The verifier then verifies that the set elements of the output (i.e., the roots of the characteristic polynomials) match the polynomial (expressed in coefficients) returned by the server. This can be efficiently done through a randomized check—see algorithm certify() from [21]. We specify that this slightly increases the communication bandwidth (the server effectively sends the output set twice, in two different encodings) but we consider this an acceptable overhead (This can be avoided by having the client perform the interpolation himself, increasing the verification time). It can also be noted that the input polynomial coefficients computation can be outsourced similarly to the server side, if the client does not have them computed already.

5.2 Experiments Setup

We now provide a comparison between TrueSet's approach and Pinocchio's approach based for set operations. For a fair comparison, we considered two different ways to construct the arithmetic circuits used by Pinocchio to verify the set operations:

- Pairwise comparison-based, which is the naive approach for performing set operations. This requires $O(n^2)$ equality comparisons.

- Sorting network-based, in which the input sets are merged and sorted first using and odd-even merge-sort network [18]. Then a check for duplicate consecutive elements is applied to include/remove repeated elements, according to the query being executed. This requires $O(n \log^2 n)$ comparator gates, and $O(n)$ equality gates.

Although the second approach is asymptotically more efficient, when translated to Pinocchio's circuits it results in numerous multiplication gates. This is due to the k-bits split gates needed to perform comparison operations, resulting into great overhead in the key generation and proof computation stages. For a k-bit possible input value, this split gate needs k multiplication constraints to constrain each bit wire to be either 0 or 1. (It should be noted that these gates translate a wire into its bit-level representation and they should not be confused with the split gates we introduce in this paper, which output the elements of a set as separate arithmetical values). On the other hand, the pairwise approach uses zero-equality gates to check for equality of elements. Each equality gate translates into only two multiplication gates, requiring only two roots.

For fairness purposes, different Pinocchio circuits were produced for each different input set cardinality we experiment with, as each wire in Pinocchio's circuits represents a single element. On the other hand, TrueSet can use the same circuit for different input cardinalities.

We consider two Pinocchio circuit implementations:

- MS Pinocchio: This is the executable built using efficient Microsoft internal libraries.
- NTL-ZM Pinocchio: This is a Pinocchio version built using exactly the same free libraries we used for our TrueSet implementation. This will help ensure having a fair comparison.

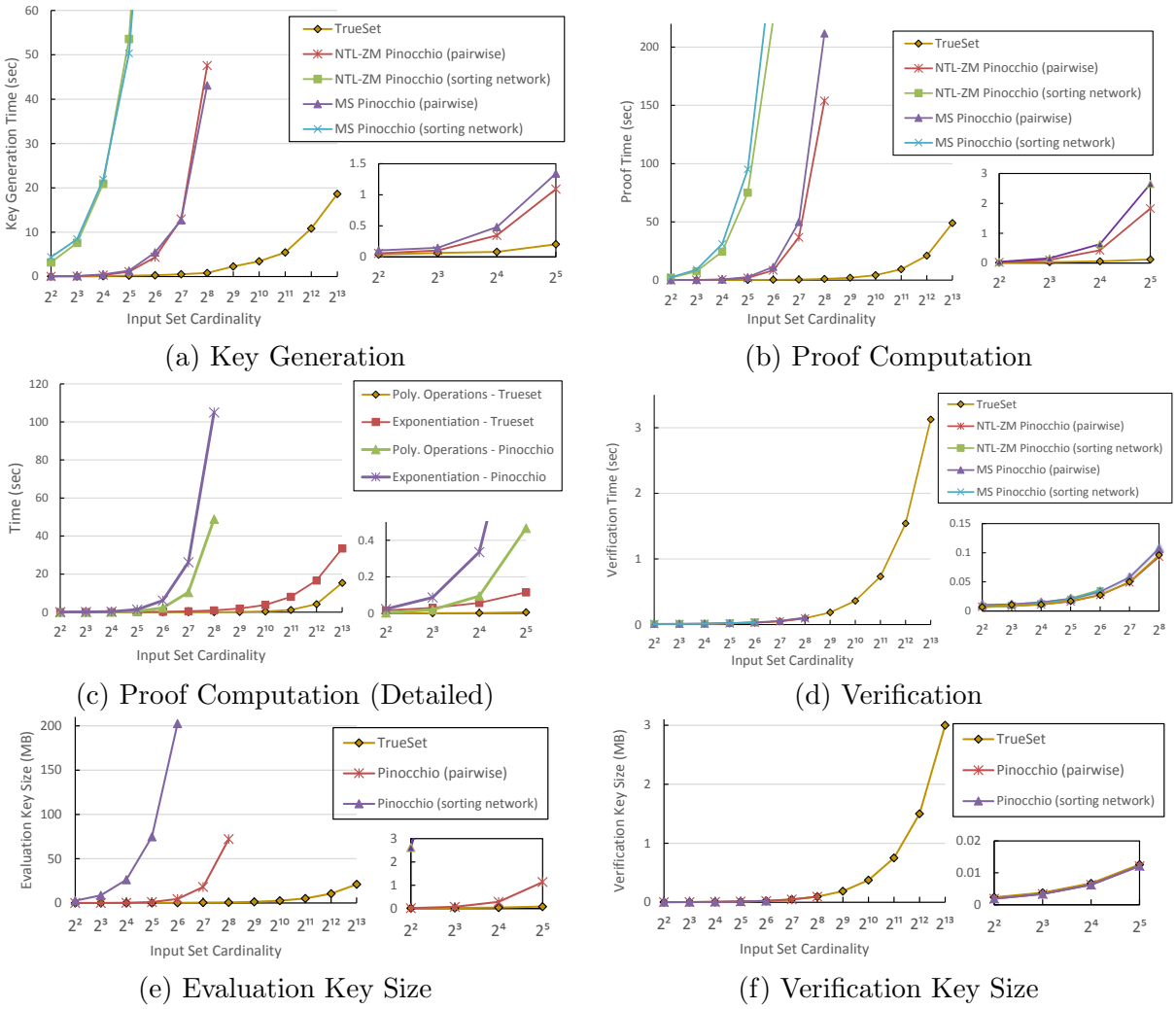
The experiments were conducted on a Lenovo IdeaPad Y580 Laptop. The executable used a single core of a 2.3 GHz Intel Core i7 with 8 GB of RAM. For the input sets, disjoint sets containing elements in $\mathbb{F}$ were assumed. For running time statistics, ten runs were collected for each data point, and the 95% confidence interval was calculated. Due to the scale of the figures, the confidence interval of the execution times (i.e., error bars) was too low to be visualized.

5.3 Single-Gate Circuit

In this subsection, we compare TrueSet and Pinocchio's protocols based on the verification of a single union operation that accepts two input sets of equal cardinalities. We study both the time overhead and the key sizes with respect to different input set cardinalities. Note that, experiments for higher input cardinalities in Pinocchio's case incur great memory overhead due to the large circuit size, therefore we were unable to even perform Pinocchio's for large input sizes.

Figure 5 shows the comparison between TrueSet's approach and Pinocchio's pairwise and sorting network approaches, versus the cardinality of each input set. The results show clearly that TrueSet outperforms both approaches in the key generation and proof computation stages by orders of magnitude, while maintaining the same verification time. Specifically, TrueSet outperforms Pinocchio in the prover's running time by *150x* when the input set cardinality is 2^8. This saving happens in both polynomial computations and exponentiation operations, as shown in Figure 5 (c). We also note that Pinocchio's pairwise comparison approach outperforms the sorting network approach due to the expensive split gates needed for comparisons in the sorting-network circuits, as discussed above, which results into a large constant affecting the performance at small cardinalities.

Considering evaluation and verification key sizes, Figure 5 also shows a comparison between TrueSet and Pinocchio under both the pairwise and sorting networks approaches. The figures demonstrate that TrueSet yields much smaller evaluation keys due to the more compact wire



(a) Key Generation

(b) Proof Computation

(c) Proof Computation (Detailed)

(d) Verification

(e) Evaluation Key Size

(f) Verification Key Size

Figure 5: Comparison between TrueSet *and Pinocchio for the case of a single union gate. In the horizontal axis, we show the cardinality of* each *input set in logarithmic scale. (Note: Each time data point is the average of ten runs. The error bars were too small to be visualized). Subfigures (a), (b) and (d) show the comparison in terms of the key generation, proof computation and verification times, while (c) shows* TrueSet*'s prover's time in more detail compared to Pinocchio's prover in the case of pairwise comparison. Subfigures (e) and (f) show the compressed evaluation and verification key sizes (The cryptographic proof for all instances is 288 bytes).*

representation it employs (a single wire for a set as opposed to a wire per element), e.g., at an input set cardinality of 2^8, the saving is about 98%. It can also be noticed that the keys generated in Pinocchio using sorting networks are much larger than the ones generated in pairwise circuits, due to the use of the split gates. On the other hand, TrueSet and Pinocchio almost maintain the same verification key sizes, as the verification key mainly depends on the number of input elements in addition to the number of output elements in the worst case. (The verification key in TrueSet is *negligibly* more than the verification key of Pinocchio, due to an additional value that is needed to be verified per each input or output set. This is because an n-element set is represented by an n-degree polynomial which requires $n + 1$ coefficients.)

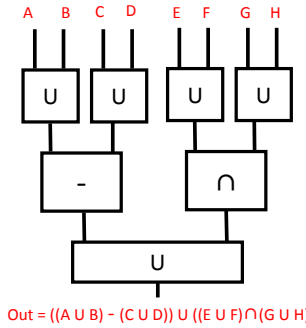


Figure 6: The multiple-gate circuit used for evaluation.

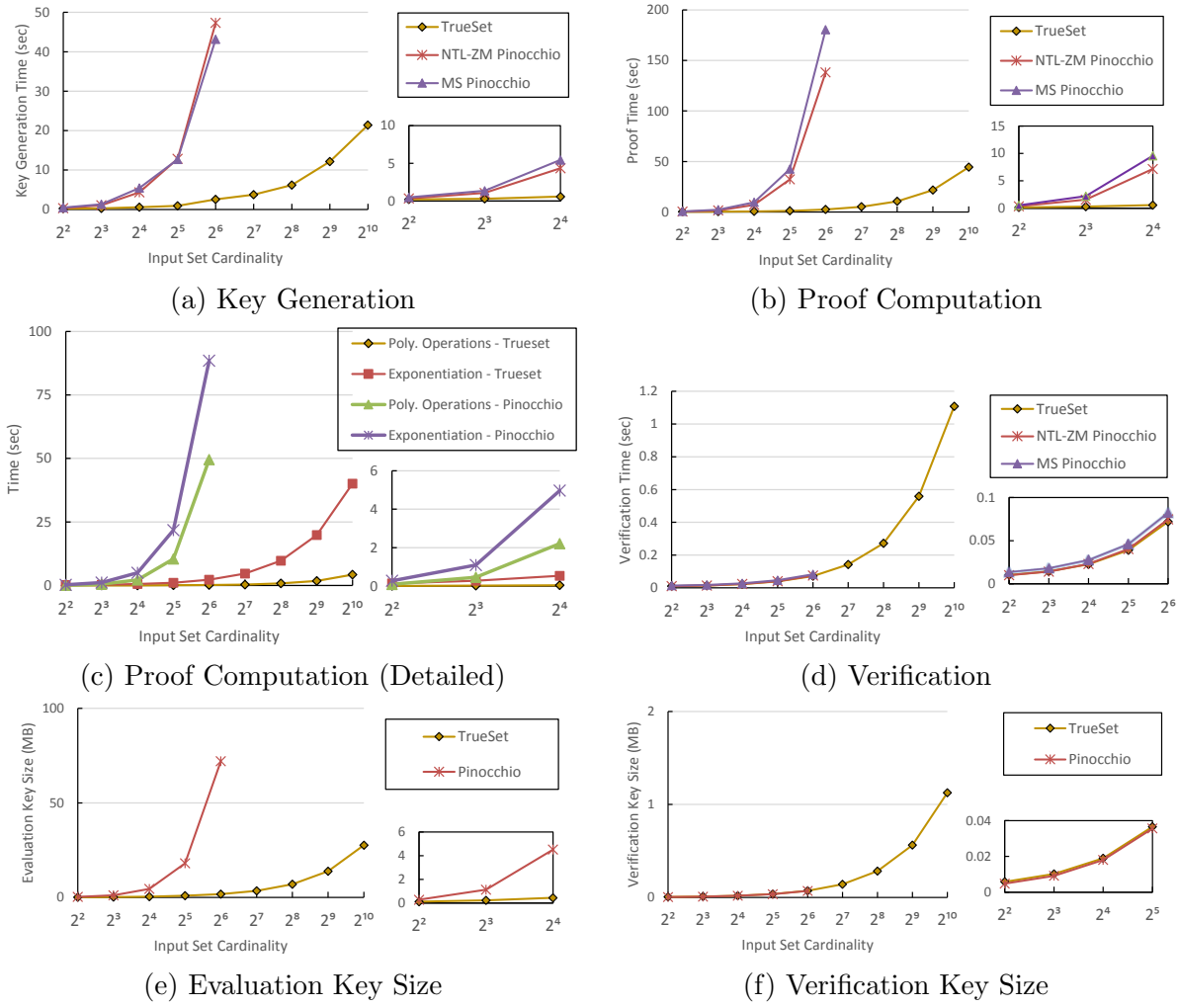


Figure 7: Comparison between TrueSet *and Pinocchio in the case of the multiple-gate circuit shown in Fig. 6, assuming the pair-wise comparison circuit for Pinocchio. In the horizontal axis, we show the cardinality of* each *input set in logarithmic scale. Subfigures (a), (b) and (d) show the comparison in terms of the key generation, proof computation and verification time, while (c) shows* TrueSet*'s prover's time in more detail compared to Pinocchio's prover time. Subfigures (e) and (f) show the compressed evaluation and verification key sizes (The cryptographic proof for all instances is 288 bytes).*

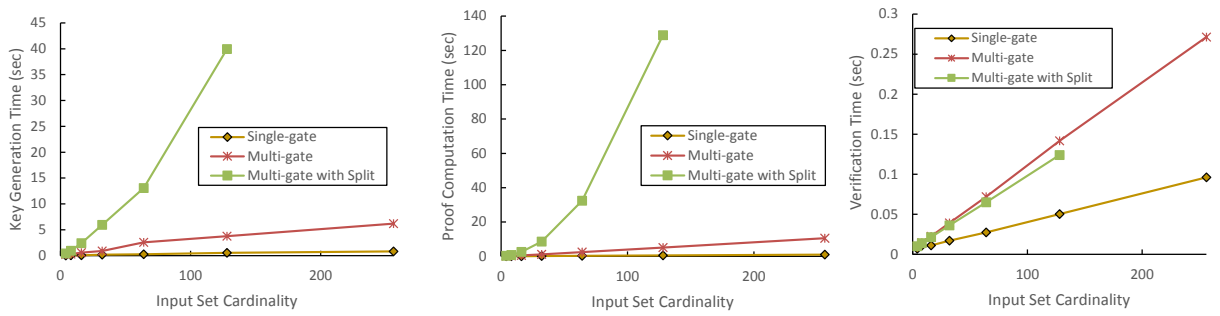


Figure 8: Summary of TrueSet *performance under all circuits in linear scale.*

	TrueSet	**MS Pinocchio**	**NTL-ZM Pinocchio**
Key Generation (sec)	13.07	43.03	47.39
Proof Computation (sec)	32.45	174.99	137.79
Verification (sec)	0.065	0.074	0.066
Evaluation Key (MB)	12.7	72.45	72.45
Verification Key (KB)	49.65	48.6	48.6

Table 1: Comparison between TrueSet *and Pinocchio on a circuit that computes* ***the cardinality and the sum*** *of the output set in the circuit in Figure 6, at input set cardinality of 64.*

5.4 Multiple-Gate Circuit

We now compare TrueSet and Pinocchio's performance for a complex set circuit consisting of multiple set operations, illustrated in Figure 6. The circuit takes eight input sets of equal cardinalities, and outputs one set. We compare both the prover's overhead and the key sizes with respect to different input set cardinalities, but this time we consider only Pinocchio circuits based on pairwise comparisons, as the sorting network approach has much larger overhead for computation times and key sizes as shown in the previous subsection.

Figure 7 shows a comparison between TrueSet's approach and Pinocchio's approach. The results again confirm that TrueSet greatly outperforms Pinocchio's elapsed time for key generation and proof computation, while maintaining the same verification time. In particular, for input set cardinality of 2^6, TrueSet's prover has a speedup of more than **50x**. In terms of key sizes, the figure confirms the observation that the evaluation key used by TrueSet is tiny compared to that of Pinocchio, e.g., 97% smaller when the input cardinality is 2^6.

5.5 Cardinality and Sum of Set Elements

Here, we evaluate TrueSet when a split gate is used to calculate the cardinality and sum for the output set of Figure 6. We compare that with Pinocchio's performance for the same functions. One important parameter that has to be defined for the split gate first is the maximum cardinality of the set it can support. This is needed for translating the split gate to the appropriate number of multiplication gates needed for verification. For example, a split gate added to the output of the circuit in Figure 6, will have to account for $4n$ set elements in the worst case, if n is the upper bound on the input set cardinalities.

Table 1 presents a comparison between TrueSet and Pinocchio in terms of the elapsed times in the three stages and the evaluation/verification key sizes, when the input set cardinality is 64. As the table shows, TrueSet can provide better performance in terms of the key generation and proof computation times (4x better proof computation time), in addition to a much smaller

public evaluation key. It can be noted that, while there definitely exists a large improvement over Pinocchio, it is not as large as the one exhibited for the previous single-gate and multiple-gate circuits. Overall, we found the split gate to be costlier than set gates since the multiplication gates introduced by the split gate increase proportionally with the number of the set elements it can support, whereas set gates are "oblivious" to the number of elements.

5.6 Discussion of Results

The evaluation of TrueSet for single-gate and multiple-gate circuits showed huge improvement for both key generation and proof computation time over Pinocchio. For example, for the single union case with 2^8-element input sets, a speed-up of 150x was obtained for the prover's time, while providing more than 98% saving in the evaluation key size. For a multiple-gate circuit comprised of seven set gates with eight input sets, each of 2^6 elements, a prover speed-up of more than 50x, and key size reduction of 97% were obtained.

As can be qualitatively inferred by our plots, these improvements in performance allow us to accommodate problem instances that are several times larger than what was considered achievable by previous works. TrueSet achieves the performance behavior that Pinocchio exhibits for sets of a few dozen elements, for sets that scale up to approximately 8000 elements, handling circuits with nearly 30x larger I/O size. Figure 8 summarizes the behavior of TrueSet for all circuits we experimented with, illustrating its performance for the three stages in linear scale. In all cases, the running time increases approximately linearly in the input size. The cost increases more abruptly when a split gate is introduced due to the added complexity discussed above. Improving the performance of the split gate is one possible direction for future work.

Remarks. We discuss here a few points related to the performance of our scheme.

Performance on Arithmetic Circuits. The presented evaluation covered the case of set circuits only, in which our construction outperformed arithmetic circuits verified using Pinocchio. Our construction can support typical arithmetic circuits as well, by assuming that the maximum polynomial degree on each wire is 0. In this case, our construction will reduce to Pinocchio's, however due to the bivariate polynomial operations, there will be more overhead in accommodating arithmetic circuits. For example, for an arithmetic circuit handling the multiplication of two 50x50 32-bit element matrices, the prover's time with TrueSet increased by 10% compared to Pinocchio.

Outsourced Sets. In the above, we assumed that the client possesses the input sets. However, it is common practice in cloud computing, to not only delegate computations but storage as well. In this case, the client initially outsources the sets to the server and then proceeds to issue set operation queries over them. This introduces the need for an additional mechanism to ensure the authenticity of the set elements used by the server. Appendix 6.6 describes a modified protocol that handles this case using Merkle tree proofs.

Supporting multisets. Finally, it should be noted that the comparisons with Pinocchio above assumed proper sets only. In a setting that accommodates multiset operations (i.e., sets that allow repetition in elements), we expect TrueSet's performance to be much better, as it can naturally handle multiset cases without adding any modifications. On the other hand, Pinocchio multiset circuits are going to become more complex due to the added complexity of taking repetitions into account. For example, in intersection gates, it will not be enough to only check that two element are equal, but it will also be necessary to make sure that the matched element was not encountered before, introducing additional overhead.

References

[1] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *CCS*, 2013.

[2] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, pages 90–108, 2013.

[3] J.-L. Beuchat, J. E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over barreto–naehrig curves. In *Pairing-Based Cryptography-Pairing 2010*, pages 21–39. Springer, 2010.

[4] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, pages 326–349, 2012.

[5] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKS and proof-carrying data. In *STOC*, pages 111–120, 2013.

[6] N. Bitansky, A. Chiesa, Y. Ishai, R. Ostrovsky, and O. Paneth. Succinct non-interactive arguments via linear interactive proofs. In *TCC*, pages 315–333, 2013.

[7] D. Boneh and X. Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *J. Cryptology*, 21(2):149–177, 2008.

[8] B. Braun, A. J. Feldman, Z. Ren, S. T. V. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, 2013.

[9] R. Canetti, O. Paneth, D. Papadopoulos, and N. Triandopoulos. Verifiable set operations over outsourced databases. In *PKC*, 2014.

[10] K.-M. Chung, Y. T. Kalai, F.-H. Liu, and R. Raz. Memory delegation. In *CRYPTO*, pages 151–168, 2011.

[11] K.-M. Chung, Y. T. Kalai, and S. P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In *CRYPTO*, pages 483–501, 2010.

[12] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, pages 465–482, 2010.

[13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, pages 626–645, 2013.

[14] T. Granlund and the GMP development team". *GMP: The GNU Multiple Precision Arithmetic Library*, 2006. Available at `http://gmplib.org/`.

[15] J. Groth. Short pairing-based non-interactive zero-knowledge arguments. In *ASIACRYPT*, pages 321–340, 2010.

[16] P. Jaccard. *Etude comparative de la distribution florale dans une portion des Alpes et du Jura.* Impr. Corbaz, 1901.

[17] L. Kissner and D. X. Song. Privacy-preserving set operations. In *CRYPTO*, pages 241–257, 2005.

[18] D. E. Knuth. *The art of computer programming.* Pearson Education, 2005.

[19] S. Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, 2000.

[20] L. Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, pages 275–292, 2005.

[21] C. Papamanthou, R. Tamassia, and N. Triandopoulos. Optimal verification of operations on dynamic sets. In *CRYPTO*, pages 91–110, 2011.

[22] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 238–252, 2013.

[23] B. Parno, M. Raykova, and V. Vaikuntanathan. How to delegate and verify in public: Verifiable computation from attribute-based encryption. In *TCC*, 2012.

[24] S. T. V. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, pages 71–84, 2013.

[25] S. T. V. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, 2012.

[26] V. Shoup. *NTL: Number theory library.* Available at `http://www.shoup.net/ntl/`.

[27] V. Shoup. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, 20(4):363–397, 1995.

[28] V. Vu, S. T. V. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE Symposium on Security and Privacy*, pages 223–237, 2013.

6 Appendix

6.1 Proof of Lemma 1

Lemma 4 *The above QPP $\mathcal{Q}$ computes $\mathcal{F}$.*

Proof: ($\Rightarrow$) Suppose $c_1(z), c_2(z), \ldots, c_N(z)$ are correct assignments of the input and output wires but there do not exist polynomials $c_{N+1}(z), \ldots, c_m(z)$ such that $\tau(x)$ divides $p(x, z)$. Then there is at least one multiplication gate r with left input x, right input y and output o, such that $p(r, z) \neq 0$. Let p be the path of multiplication gates that contains multiplication gate r starting from an input polynomial $c_i(z)$ to an output polynomial $c_j(z)$, where $i, j \leq N$. Since $c_i(z)$ and $c_j(z)$ are correct assignments, there must exist polynomials $c_x(z)$ and $c_y(z)$ such that $c_x(z)c_y(z) = c_o(z)$. Since r has a single left input, a single right input and a single output it holds $v_x(r) = 1$ and $v_i(r) = 0$ for all $i \neq x$. Similarly, $w_y(r) = 1$ and $w_i(r) = 0$ for all $i \neq y$ and $y_o(r) = 1$ and $y_i(r) = 0$ for all $i \neq o$. Therefore $p(r, z) \neq 0$ implies that for all polynomials $c_x(z)$, $c_y(z)$, $c_o(z)$, it is $c_x(z)c_y(z) \neq c_o(z)$, a contradiction.

($\Leftarrow$) Suppose $\tau(x)$ divides $p(x, z)$. Then $p(r, x) = 0$ for all multiplication gates r. By the definition of the polynomials $v_i(x)$, $w_i(x)$, $y_i(x)$, it follows that $c_1(z), c_2(z), \ldots, c_m(z)$ are correct assignments on the circuit wires. ■

6.2 Computational Assumptions

Assumption 1 (q-PDH assumption [15]) *The q-power Diffie-Hellman (q-PDH) assumption holds for $\mathcal{G}$ if for all $\mathcal{A}$ we have*

$$\Pr\left[\begin{array}{l}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow\mathcal{G}(1^k);s\leftarrow\mathbb{Z}_p^*;\\ \sigma\leftarrow(p,\mathbb{G},\mathbb{G}_T,e,\mathbf{G});\\ y\leftarrow\mathcal{A}(\sigma):y=g^{s^{q+1}}.\end{array}\right]=\mathsf{neg}(k)\,.$$

where

$$\mathbf{G}=\left[g,g^s,\ldots,g^{s^q},g^{s^{q+2}},\ldots,g^{s^{2q}}\right]\,.$$

Assumption 2 (q-PKE assumption [15]) *The q-power knowledge of exponent assumption holds for $\mathcal{G}$ if for all $\mathcal{A}$ there exists a non-uniform probabilistic polynomial time extractor $\chi_{\mathcal{A}}$ such that*

$$\Pr\left[\begin{array}{l}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow\mathcal{G}(1^k);\{\alpha,s\}\leftarrow\mathbb{Z}_p^*;\\ \sigma\leftarrow(p,\mathbb{G},\mathbb{G}_T,e,\mathbf{G});\\ (c,\hat{c};a_0,a_1,\ldots,a_q)\leftarrow(\mathcal{A}||\chi_{\mathcal{A}})(\sigma,z):\\ \hat{c}=c^\alpha\wedge c\neq g^{\prod_{i=0}^q a_i s^i}.\end{array}\right]=\mathsf{neg}(k)$$

for any auxiliary information $z\in\{0,1\}^{\mathsf{poly}(k)}$ that is generated independently of α and where

$$\mathbf{G}=\left[g,g^s,\ldots,g^{s^q},g^\alpha,g^{\alpha s},\ldots,g^{\alpha s^q}\right]\,.$$

Note that $(y;z)\leftarrow(\mathcal{A}||\chi_{\mathcal{A}})(x)$ signifies that on input x, $\mathcal{A}$ outputs y, and that $\chi_{\mathcal{A}}$, given the same input x and $\mathcal{A}$'s random tape, produces z.

Assumption 3 (q-SDH assumption [7]) *The q-strong Diffie-Hellman (q-SDH) assumption holds for $\mathcal{G}$ if for all $\mathcal{A}$ we have*

$$\Pr\left[\begin{array}{l}(p,\mathbb{G},\mathbb{G}_T,e,g)\leftarrow\mathcal{G}(1^k);\{s\}\leftarrow\mathbb{Z}_p^*;\\ \sigma\leftarrow(p,\mathbb{G},\mathbb{G}_T,e,\mathbf{G});\\ (y,c)\leftarrow\mathcal{A}(\sigma):y=e(g,g)^{\frac{1}{s+c}}.\end{array}\right]=\mathsf{neg}(k)\,.$$

where

$$\mathbf{G}=\left[g,g^s,\ldots,g^{s^q}\right]\,.$$

6.3 Succinct Non-Interactive Arguments of Knowledge (SNARKs)

Definition 6 (SNARK) *Algorithms* $(\mathsf{KeyGen},\mathsf{Prove},\mathsf{Verify})$ *give a* succinct non-interactive argument of knowledge (SNARK) *for an NP language L with corresponding NP relation R_L if:*

Completeness: *For all $x\in L$ with witness $w\in R_L(x)$, the probability:*

$$\Pr\left[\mathsf{Verify}(\mathsf{sk},x,\pi)=0\;\middle|\;\begin{array}{c}(\mathsf{pk},\mathsf{sk})\leftarrow\mathsf{KeyGen}(1^k),\\ \pi\leftarrow\mathsf{Prove}(\mathsf{pk},x,w)\end{array}\right]$$

is a negligible function of k.

Adaptive soundness: *For any probabilistic polynomial-time algorithm $\mathcal{A}$, the probability:*

$$\Pr\left[\begin{array}{c}\mathsf{Verify}(\mathsf{sk},x,\pi)=1\\ \wedge\;(x\notin L)\end{array}\;\middle|\;\begin{array}{c}(\mathsf{pk},\mathsf{sk})\leftarrow\mathsf{KeyGen}(1^k),\\ (x,\pi)\leftarrow\mathcal{A}(1^k,\mathsf{pk})\end{array}\right]$$

is negligible in k.

Succinctness: *The length of a proof is given by* $|\pi| = \mathsf{poly}(k)\mathsf{poly}\log(|x| + |w|)$.

Extractability: *For any poly-size prover* Prv*, there exists an extractor* $\mathsf{Extract}$ *such that for any statement* x*, auxiliary information* μ*, the following holds:*

$$\Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{sk}) \leftarrow \mathsf{KeyGen}(1^k) \\ \pi \leftarrow \mathsf{Prv}(\mathsf{pk}, x, \mu) \\ \mathsf{Verify}(\mathsf{sk}, x, \pi) = 1\end{array} \wedge \begin{array}{c} w \leftarrow \mathsf{Extract}(\mathsf{pk}, \mathsf{sk}, x, \pi) \\ w \notin R_L(x)\end{array}\right] = \mathsf{negl}(k)\,.$$

Zero-knowledge: *There exists a simulator* Sim*, such that for any probabilistic polynomial-time adversary* $\mathcal{A}$*, the following holds:*

$$\Pr\left[\begin{array}{c}\mathsf{pk} \leftarrow \mathsf{KeyGen}(1^k); (x,w) \leftarrow \mathcal{A}(\mathsf{pk}); \\ \pi \leftarrow \mathsf{Prove}(\mathsf{pk}, x, w) : (x,w) \in R_L \\ \text{and } \mathcal{A}(\pi) = 1\end{array}\right] \simeq \Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{state}) \leftarrow \mathsf{Sim}(1^k); (x,w) \leftarrow \mathcal{A}(\mathsf{pk}); \\ \pi \leftarrow \mathsf{Sim}(\mathsf{pk}, x, \mathsf{state}) : (x,w) \in R_L \\ \text{and } \mathcal{A}(\pi) = 1\,.\end{array}\right]$$

We say that a SNARK is *publicly verifiable* if $\mathsf{sk} = \mathsf{pk}$. In this case, proofs can be verified by anyone with pk. Otherwise, we call it a *secretly-verifiable* SNARK, in which case only the party with sk can verify.

6.4 Verifiable Computation

We now define Verifiable Computation.

Definition 7 (Public Verifiable Computation [22]) *A public verifiable computation scheme consists of a set of three polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Compute}, \mathsf{Verify})$ *defined as follows.*

1. $\{\mathsf{EK}_f, \mathsf{VK}_f\} \leftarrow \mathsf{KeyGen}(f, 1^k)$*. The randomized key generation algorithm takes the function* f *to be outsourced and security parameter* k*; it outputs a public evaluation key* EK_f*, and a public verification key* VK_f*.*

2. $(y, \pi_y) \leftarrow \mathsf{Compute}(\mathsf{EK}_f, u)$*: The deterministic worker algorithm uses the public evaluation key* EK_f *and input* u*. It outputs* $y = f(u)$ *and a proof* π_y *of* y*'s correctness.*

3. $\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{VK}_f, u, y, \pi_y)$*: Given the verification key* VK_f*, the deterministic verification algorithm outputs 1 if* $f(u) = y$*, and 0 otherwise.*

4. ***Correctness****. For any function* f*, and any input* u *to* f*, if we run* $\{\mathsf{EK}_f, \mathsf{VK}_f\} \leftarrow \mathsf{KeyGen}(f, 1^k)$ *and* $(y, \pi_y) \leftarrow \mathsf{Compute}(\mathsf{EK}_f, u)$*, then we always get* $1 \leftarrow \mathsf{Verify}(\mathsf{VK}_f, u, y, \pi_y)$*.*

5. ***Security****. For any function* f *and any probabilistic polynomial-time adversary* $\mathcal{A}$*, the probability* $\Pr[(u', y', \pi'_y) \leftarrow \mathcal{A}(\mathsf{EK}_f, \mathsf{VK}_f) : f(u') \neq y' \wedge 1 \leftarrow \mathsf{Verify}(\mathsf{VK}_f, u', y', \pi'_y)]$ *is* $\mathsf{neg}(k)$*.*

6. ***Efficiency****.* KeyGen *is assumed to be a one-time operation whose cost is amortized over many calculations, but we require that* Verify *is cheaper than evaluating* f*.*

6.5 Proof of Theorem 1

We now give proofs of completeness, soundness, extractability and zero-knowledge.

Completeness. Follows by inspection.

Soundness. Assume that there exists an adversary $\mathcal{A}$ who returns a cheating proof. Then we show how to construct an adversary $\mathcal{B}$ that uses $\mathcal{A}$ and a $2(n+1)q$-PKE assumption knowledge extractor

and breaks either the $(n+1)q$-PDH assumption or the $2(n+1)q$-SDH assumption, where $q = 4d+4$. We show that $\mathcal{B}$'s attack is successful with probability $1/2$. First, $\mathcal{B}$ flips a coin $b \in \{0,1\}$. If $b = 0$, $\mathcal{B}$ asks for a challenge of an $(n+1)q$-PDH assumption, else $\mathcal{B}$ asks for a challenge of a $2(n+1)q$-SDH assumption.

$$\mathbf{G}' = \begin{bmatrix} g^s & \dots & g^{s^q} & g^{s^{q+2}} & \dots & g^{s^{2q}} \\ g^{s^{2q+1}} & \dots & g^{s^{3q}} & g^{s^{3q+2}} & \dots & g^{s^{4q}} \\ g^{s^{4q+1}} & \dots & g^{s^{5q}} & g^{s^{5q+2}} & \dots & g^{s^{6q}} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ g^{s^{2nq+1}} & \dots & g^{s^{(2n+1)q}} & g^{s^{(2n+1)q+2}} & \dots & g^{s^{2(n+1)q}} \end{bmatrix}.$$

The challenge above is a subset of an $(n+1)q$-PDH instance (Assumption 1), since more elements than one are missing.

Case $b = 1$. $\mathcal{B}$ is given a $2(n+1)q$-SDH instance

$$\boldsymbol{G} = \left[g, g^s, \dots, g^{s^{2(n+1)q}}\right].$$

Generating keys. The adversary $\mathcal{A}$ generates a function with N inputs/outputs that has QPP $(\tau(x), \mathcal{V}, \mathcal{W}, \mathcal{Y})$ of size $m = N + d$ and degree d. Recall that $I_m = \{N+1, \dots, m\}$ are the non-IO-related indices. If $b = 0$, $\mathcal{B}$ sets $t = s^{2q}$, otherwise it set $t = Rs^{2q}$ for random R and provides a carefully generated evaluation key to $\mathcal{A}$, i.e.,

- $\{g_v^{t^i v_k(s)}\}_{(i,k)\in[n]\times I_m}$, $\{g_w^{t^i w_k(s)}\}_{(i,k)\in[n]\times I_m}$, $\{g_y^{t^i y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
- $\{g_v^{t^i \alpha_v v_k(s)}\}_{(i,k)\in[n]\times I_m}$, $\{g_w^{t^i \alpha_w w_k(s)}\}_{(i,k)\in[n]\times I_m}$, $\{g_y^{t^i \alpha_y y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
- $\{g_v^{t^i \beta\cdot v_k(s)} g_w^{t^i \beta\cdot w_k(s)} g_y^{t^i \beta\cdot y_k(s)}\}_{(i,k)\in[n]\times I_m}$.
- $\{g^{t^i s^j}\}_{(i,j)\in[2n]\times[d]}$.

where $g_v = g^{r'_v s^{d+1}}$, $g_w = g^{r'_w s^{2(d+1)}}$ and $g_y = g^{r'_y s^{3(d+1)}}$, where α_v, α_w, α_y, r'_v and r'_w are chosen uniformly at random and $r'_y = r'_v r'_w$.

We now describe how $\mathcal{B}$ chooses β and γ in the challenge above. If $b = 1$, both β and γ are picked uniformly at random. Otherwise $\mathcal{B}$ sets

$$\beta = s^{q-(4d+3)}\beta(s),$$

where $\beta(x)$ is a polynomial of degree at most $3d+3$ sampled uniformly at random such that the polynomial

$$\beta(x)[r'_v v_k(x) + r'_w x^{d+1} w_k(x) + r'_y x^{2(d+1)} y_k(x)]$$

has a zero coefficient in front of x^{3d+3} for all $k = 1, \dots, m$ (see Lemma 10 of [13]). It is easy now to see that, with such a choice of β, the term $g_v^{t^i \beta\cdot v_k(s)} g_w^{t^i \beta\cdot w_k(s)} g_y^{t^i \beta\cdot y_k(s)}$, for all k, can be written as

$$\begin{aligned} & g_v^{t^i \beta\cdot v_k(s)} g_w^{t^i \beta\cdot w_k(s)} g_y^{t^i \beta\cdot y_k(s)} \\ = \; & g^{t^i [s^{q-(4d+3)}\beta(s)[r'_v s^{d+1} v_k(s) + r'_w s^{2(d+1)} w_k(s) + r'_y s^{3(d+1)} y_k(s)]]} \\ = \; & g^{t^i [s^{q-(3d+2)}\beta(s)[r'_v v_k(s) + r'_w s^{d+1} w_k(s) + r'_y s^{2(d+1)} y_k(s)]]} \end{aligned} \tag{6.2}$$

where its exponent has a zero term in front of $t^i s^{q+1} = s^{i(2q)+q+1} = s^{q(2i+1)+1}$ and the maximum degree in variable x is less than $2(n+1)q$. Therefore $\mathcal{B}$ can efficiently generate the term $g_v^{t^i\beta\cdot v_k(s)} g_w^{t^i\beta\cdot w_k(s)} g_y^{t^i\beta\cdot y_k(s)}$ from its challenge.

To choose γ, $\mathcal{B}$ generates γ' uniformly at random and sets $\gamma = \gamma' s^{q+2}$. It easy to see that both g^γ and $g^{\beta\gamma}$ can be efficiently generated from the challenge.

Extracting the exponents. Let now $\gamma_v, \gamma_w, \gamma_y, \gamma_h, \kappa_v, \kappa_w, \kappa_y, \Lambda$ be a cheating proof. Since in the verification we have

1. $e(\gamma_v, g^{\alpha_v}) = e(\kappa_v, g)$.
2. $e(\gamma_w, g^{\alpha_w}) = e(\kappa_w, g)$.
3. $e(\gamma_y, g^{\alpha_y}) = e(\kappa_y, g)$.

we can use a subset of the $(2n+2)q$-PKE assumption to extract the polynomials $v_m(x)$, $w_m(x)$ and $y_m(x)$. Note, that, due to the way $\mathcal{B}$ picked $t = s^{2q}$ ($b = 0$) in these polynomials, the expression $x^y = x^{i(2q)+j}$ for $(i,j) \in [n] \times [d]$ is mapped to

$$x^j \times z^i \,.$$

For $b = 1$, it is easy to see that the mapping is

$$\frac{x^j \times z^i}{R} \,.$$

Consistency. Let us suppose now that there do not exist polynomials $c_k(z)$ such that $v_m(x,z) = \sum_{k\in I_m} c_k(z) v_k(x)$, $w_m(x,z) = \sum_{k\in I_m} c_k(z) w_k(x)$ and $y_m(x,z) = \sum_{k\in I_m} c_k(z) y_k(x)$. Due to verification, $e(\lambda_v \cdot \lambda_w \cdot \lambda_y, g^{\beta\gamma}) = e(\Lambda, g^\gamma)$ hence $e(g^{r'_v s^{d+1} v_m(s,t)} g^{r'_w s^{2(d+1)} w_m(s,t)} g^{r'_y s^{3(d+1)} y_m(s,t)}, g^{\beta\gamma}) = e(\Lambda, g^\gamma)$, and it follows that

$$\begin{aligned} \Lambda &= g^{\beta(r'_v s^{d+1} v_m(s,t) + r'_w s^{2(d+1)} w_m(s,t) + r'_y s^{3(d+1)} y_m(s,t))} \\ &= g^{s^{q-(4d+3)}\beta(s)(r'_v s^{d+1} v_m(s,t) + r'_w s^{2(d+1)} w_m(s,t) + r'_y s^{3(d+1)} y_m(s,t))} \,, \end{aligned}$$

where $\beta(x)$ is the polynomial such that any polynomial that is produced as a linear combination of

$$\left\{ x^{q-(4d+3)} \beta(x) (r'_v x^{(d+1)} v_k(x) + r'_w x^{2(d+1)} w_k(x) + r'_y x^{3(d+1)} y_k(x)) \right\}_{i=1,\ldots,k}$$

has a zero term in front of x^{q+1}. Polynomials that are not produced as a linear combination of these values have a non-zero term in front of x^{q+1} that is distributed uniformly at random (when sampling $b(x)$ uniformly at random based on the above restriction).

Since we assumed there does not exist polynomials $c_k(z)$ such that $v_m(x,z) = \sum_{k\in I_m} c_k(z) v_k(x)$, $w_m(x,z) = \sum_{k\in I_m} c_k(z) w_k(x)$ and $y_m(x,z) = \sum_{k\in I_m} c_k(z) y_k(x)$, it follows that the exponent of Λ has a non-zero term in front of $t^i x^{q+1}$ for some i. It is easy to subtract the remaining terms from Λ and produce $g^{t^i x^{q+1}} = g^{x^{2iq+q+1}}$, for some i. If $b = 0$, $\mathcal{B}$ breaks a subset of $(n+1)q$-PDH assumption and therefore it breaks the $(n+1)q$-PDH assumption. Otherwise it aborts.

Divisibility. Let now

$$\lambda_v = g_v^{v_0(s,t)},\ \lambda_w = g_w^{w_0(s,t)} \text{ and } \lambda_y = g_y^{y_0(s,t)} \,,$$

as computed by the verification algorithm, with polynomials $v_0(x,z) = \sum_{k\in[N]} c_k(z)v_k(x)$, $w_0(x,z) = \sum_{k\in[N]} c_k(z)w_k(x)$ and $y_0(x,z) = \sum_{k\in[N]} c_k(z)y_k(x)$. Let us define now the polynomial $p(x,z) = v(x,z)w(x,z) - y(x,z)$. This can be written as

$$p(x,z) = \left(\sum_{k=1}^{m} c_k(z)v_k(x)\right)\left(\sum_{k=1}^{m} c_k(z)w_k(x)\right) - \left(\sum_{k=1}^{m} c_k(z)y_k(x)\right).$$

Suppose it is not divided by $\tau(x) = \prod_{i=1}^{d}(x - r_i)$ (equivalently, at least one of the characteristic polynomials $c_k(z)$ used by the adversary is not the honestly computed one). Then there exists a factor r_u of $\tau(x)$ that does not divide $p(x,z)$ which can be efficiently computed by dividing $p(x,z)$ with all factors of $\tau(x)$. Therefore $p(x,z)$ can be written as $p(x,z) = r(x,z)(x - r_u) + \kappa(z)$. But the divisibility requirement holds:

$$\begin{aligned} e(\lambda_v \cdot \gamma_v, \lambda_w \cdot \gamma_w)/e(\lambda_y \cdot \gamma_y, g) &= e(\gamma_h, g^{\tau(s)}) \\ e(g^{\sum_{k=1}^{m} c_k(t)v_k(s)}, g^{\sum_{k=1}^{m} c_k(t)w_k(s)}) \cdot e(g^{-\sum_{k=1}^{m} c_k(t)y_k(s)}, g) &= e(\gamma_h, g^{\tau(s)}) \\ e(g,g)^{r(s,t)(s-r_u)+\kappa(t)} &= e(\gamma_h, g^{\tau(s)}). \end{aligned}$$

If $b = 0$, $\mathcal{B}$ aborts. Otherwise, when $b = 1$, it is $z = Rx^{2q}$. Since $\kappa(z)$ has degree n, it has at most n roots. Let w be a root of $\kappa(z)$. Note now that $\kappa(z)$ is divided by $(x - r_u)$ iff $\kappa(Rx^{2q})$ is divided by $(x - r_u)$ iff $Rr_u^{2q} = w$, where w is a root of $\kappa(z)$. Since R is random, the probability of that event is negligible. We can thus write $\kappa(Rx^{2q}) = \pi(x)(x - r_u) + \lambda$, where λ is a constant. Therefore we have

$$e(g,g)^{r(s,t)(s-r_u)+\kappa(t)} = e(\gamma_h, g^{\prod_{i\neq u}(s-r_i)}) \Leftrightarrow e(g,g)^{r(s,t)(s-r_u)+\pi(s)(s-r_u)+\lambda} = e(\gamma_h, g^{\tau(s)}),$$

which gives

$$e(g,g)^{\frac{1}{s-r_u}} = \left(e(\gamma_h, P)e(g,g)^{-r(s,t)(s-r_u)-\pi(s)}\right)^{\lambda^{-1}},$$

where $P = g^{\prod_{i\neq u}(s-r_i)}$, breaking the $2(n+1)q$-SDH assumption.

Extractability. In the soundness proof we show how to extract the coefficients of the polynomial $v_m(x,z) = \sum_{k\in I_m} c_k(z)v_k(x)$, but not the actual polynomials $c_k(z)$. Here we show how to extract the actual polynomials $c_k(z)$ for $k \in I_m$. Write the extracted polynomial as

$$v_m(z,x) = a_d(z)x^D + a_d(z)x^{D-1} + \ldots + a_1(z)x + a_0(z)$$

for some $D < d$, since $|I_m| < d$. Let now $v_k(x) = b_{kd}x^d + b_{k(d-1)}x^{d-1} + \ldots + b_{k0}$, for $k \in I_m$. It is easy to see that the following system of linear equations need to hold:

$$\begin{bmatrix} b_{10} & b_{20} & \ldots & b_{D0} \\ b_{11}x & b_{21}x & \ldots & b_{D1}x \\ & \ldots & & \\ b_{1d}x^d & b_{2d}x^d & \ldots & b_{Dd}x^d \end{bmatrix} \cdot \begin{bmatrix} c_1(z) \\ c_2(z) \\ \ldots \\ \ldots \\ \ldots \\ c_D(z) \end{bmatrix} = \begin{bmatrix} a_0(z) \\ a_1(z) \\ \ldots \\ a_d(z) \end{bmatrix}.$$

Note that the above system has a solution since the multiplying matrix contains columns that are linearly independent. To see that, suppose not. Then there exist non-zero $w_1, w_2, \ldots, w_D$ such that $\sum_{i=1}^{D} w_i v_i(x) = 0$. This means that $\sum_{i=1}^{D} w_i v_i(x) = 0$. Note however this is not true since we know for $x = r_g$ there exists a k such that $v_k(r_g) = 1$.

Zero-knowledge. For achieving zero-knowledge, the prover computes the following proof

- $g_v^{v_m(s,t)+\delta_v t(s)}$, $g_w^{w_m(s,t)+\delta_w t(s)}$, $g_y^{y_m(s,t)+\delta_y t(s)}$, $g^{h(s,t)}$
- $g_v^{\alpha_v(v_m(s,t)+\delta_v t(s))}$, $g_w^{\alpha_w(w_m(s,t)+\delta_w t(s))}$, $g_y^{\alpha_y(y_m(s,t)+\delta_y t(s))}$
- $g_v^{\beta\cdot(v_m(s,t)+\delta_v t(s))} g_w^{\beta\cdot(w_m(s,t)+\delta_w t(s))} g_y^{\beta\cdot(y_m(s,t)+\delta_y t(s))}$

where δ_v, δ_w and δ_y are picked uniformly at random. Therefore the proof elements become completely randomized, without loosing structure. To achieve that, in the public key pk we also include the terms $g_v^{t(s)}, g_w^{t(s)}, g_y^{t(s)}$, $g_v^{\alpha_v t(s)}, g_w^{\alpha_w t(s)}, g_y^{\alpha_y t(s)}$ and $g_v^{\beta t(s)}, g_w^{\beta t(s)}, g_y^{\beta t(s)}$.

6.6 Verifiable Computation for Circuits on Sets

In this section, we present our verifiable computation (VC) schemes for set circuits (for the definition of VC, see Section 6.4 in the Appendix). First, observe that our SNARK construction for set circuits immediately yields a VC scheme for set circuits where the client sends to the server the input upon which the circuit is to be evaluated. Given a set circuit $\mathcal{C}$, one can compute a corresponding SNARK as described in Section 4.2 and send the evaluation key along with $\mathcal{C}$ to the server. Consequently upon sending a number of sets $X_1, \ldots, X_{N-1}$ he can receive answer set O and proof that $\{X_1, \ldots, X_{N-1}, O\} \in \mathcal{L}(\mathcal{C})$. The client can benefit from the re-usability of our SNARK, as well as from the fact that it is publicly verifiable hence multiple clients can ask queries. The verification cost for the client is of course $O(\sum_{i=1}^{N-1} |X_i| + |O|)$.

Consider now the scenario where a client commits to m sets $X_1, X_2, \ldots, X_m$, outsources the sets to the server and every time he would like to evaluate the set circuit on a subset of $N-1$ sets of his liking. We present a VC construction for this problem that combines verifiable computation with outsourcing of data. Indeed, once the owner of sets X_i runs a setup phase and outsources the sets to the server with a corresponding circuit $\mathcal{C}$, any client with access to public information can issue queries over an arbitrary subset of them without ever seeing X_i's (just a succinct digest of them), making this approach ideal for multi-client environments where storage is costly.

Each set X_i with cardinality n_i is originally represented by its *accumulation value* $a_i = g^{\prod_{j=1}^{n_i}(t-x_j)}$ (similar to the bilinear accumulator of [20]) where the exponent is the characteristic polynomial of X_i evaluated for $z = t$ and consequently a Merkle tree is deployed over the m values a_i. The evaluation key contains SNARK-related elements and the Merkle tree, and the verification key contains also the tree's digest d. At a high level, the function f that the client outsources corresponding to set circuit $\mathcal{C}$, takes us input a set of $N-1$ indices from $[1, \ldots, m]$ that correspond to the sets to be used as inputs of $\mathcal{C}$, and returns the circuit output. That is, given a collection of sets $\mathcal{X} = \{X_1, \ldots, X_m\}$, the function $f_{\mathcal{C},\mathcal{X}}$ corresponding to circuit $\mathcal{C}$ with $N-1$ input wires, is $f(i_1, \ldots, i_{N-1}) = \mathcal{C}(X_{i_1}, \ldots, X_{i_{N-1}})$. For the evaluation of $\mathcal{C}$ the server also uses as witness the sets X_i (the ones that correspond to the requested indices). Our concrete VC construction is as follows:

$\underline{\{\mathsf{EK}_f, \mathsf{VK}_f\} \leftarrow \mathsf{KeyGen}(f_{\mathcal{X},\mathcal{C}}, \mathcal{X}, \mathcal{C}, 1^k)}$: Run the SNARK key generation $\mathsf{KeyGen}(\mathcal{C}, 1^k)$ to receive corresponding $(\mathsf{pk}, \mathsf{sk})$. For each set X_i compute $a_i = g^{\prod_{j=1}^{n_i}(t-x_j)}$. Choose appropriate collision-resistant hash function h and compute a Merkle tree $\mathcal{M}$ over values (i, a_i) with digest d. Output

$$\{\mathsf{EK}_f := (\mathsf{pk}, a_1, \ldots, a_n, \mathcal{M}), \mathsf{VK}_f := (\mathsf{pk}, d, h)\}.$$

$\underline{\{y, \pi_y\} \leftarrow \mathsf{Compute}(\mathsf{EK}_f, i_1, \ldots, i_{N-1})}$: Compute $y = \mathcal{C}(X_{i_1}, \ldots, X_{i_{N-1}})$. Execute the SNARK prover algorithm $\mathsf{Prove}(\mathsf{pk}, (X_{i_1}, \ldots, X_{i_{N-1}}, y), w)$ to receive π (w is the corresponding inner wire values). Let X_j for $j = 1, \ldots, N-1$ denote the sets corresponding to indices $i_1, \ldots, i_{N-1}$. For each corresponding value a_j compute a Merkle tree proof p_j for tree $\mathcal{M}$. For each input set X_j, compute value

$\delta_j = g^{\alpha_v \prod_{l=1}^{n_j}(t-x_l)}$. Finally, compute values $\lambda_v^{(i)} = g^{c_i(t)v_i(s)}$, $\lambda_w^{(i)} = g^{c_i(t)w_i(s)}$ and $\lambda_y^{(i)} = g^{c_i(t)y_i(s)}$ for $k \in [N]$. Output

$$\{y, \pi_y = (\pi, a_1, \ldots, a_{N-1}, \delta_1, \ldots, \delta_{N-1}, p_1, \ldots, p_{N-1}, \lambda_v^{(1)}, \lambda_w^{(1)}, \lambda_y^{(1)}, \ldots, \lambda_v^{(N)}, \lambda_w^{(N)}, \lambda_y^{(N)})\}\,.$$

$\{0,1\} \leftarrow \mathsf{Verify}(\mathsf{VK}_f, i_1, \ldots, i_{N-1}, y, \pi_y)$: For each value a_j verify its validity w.r.t d using proof p_j. If it fails for any of them, output 0 and halt. For each pair a_j, δ_j check the equality $e(a_j, g^{\alpha_v}) = e(\delta_j, g)$. If any of them fails, output 0 and halt. Compute $a_N := g^{\prod_{l=1}^{|y|}(t-x_l)}$.
For $j \in [N]$ check:

$$e(\lambda_v^{(j)}, g) \stackrel{?}{=} e(a_j, g^{v_j(s)}) \wedge e(\lambda_w^{(j)}, g) \stackrel{?}{=} e(a_j, g^{w_j(s)}) \wedge e(\lambda_y^{(j)}, g) \stackrel{?}{=} e(a_j, g^{y_j(s)})\,.$$

(recall that values $g^{v_j(s)}$, etc. are included in the pk). If any of them fails, output 0 and halt. Compute value $\lambda_v = \prod_{k\in[N]} \lambda_v^{(k)}$ and likewise for values λ_w, λ_y. Finally, run the SNARK verification algorithm (see Section 3) using the computed λ values. If it accepts, output 1 otherwise 0.

Theorem 3 (VC for outsourced sets) *Let $\mathcal{C}$ be a set circuit that has d total gates and N total inputs and outputs. Let n be an upper bound on the cardinalities of the sets on the wires and let $q = 16d_1 + 20d_2 + 4$, where d_1 is the number of intersection gates and d_2 is the number of union and difference gates ($d = d_1 + d_2$). Let $\mathcal{X} = \{X_1, \ldots, X_m\}$ be a collection of m sets, with $M = \sum_{i=1}^m |X_i|$ and $\mu = \max_{i=1}^m\{|X_i|\}$. Finally, let $K = \max\{2n+1, \mu\}$. The scheme $\{\mathsf{KeyGen}, \mathsf{Compute}, \mathsf{Verify}\}$ is a verifiable computation scheme for function $f_{\mathcal{C},\mathcal{X}}$ such that: (1) it is* secure *under the Kq-PKE, the $(n+1)q$-PDH, the Kq-SDH assumptions, and the existence of CRH functions; (2) Algorithm* KeyGen *runs in $O(dn+M)$ time; (3) Algorithm* $\mathsf{Compute}$ *runs in $O(d\nu \log^2 \nu \log\log \nu + (N-1)\log m)$ time, where ν is the maximum cardinality of the sets on the wires; (4) Algorithm* Verify *runs in $O(\sum_{i\in[N]} n_i)$ time, where n_i is the cardinality of the set on wire i; (5) Proofs consist of $O(N)$ ($5N+6$ in practice) group elements and $O((N-1)\log m)$ hash values.*

Proof Sketch: First observe that a cheating prover can cheat either by using sets with different accumulation values, different sets with the same accumulation values, or cheat in the SNARK proof construction. By the security of the Merkle tree, the first case can happen with negligible-probability, or a collision for function h is found. Assuming the second happens, the used sets can be extracted by the q-PKE assumption. Let X be one of the originally accumulated sets and X' the extracted one, such that $X \neq X'$ and

$$g^{\prod_{l=1}^{|X|}(t-x_l)} = a = g^{\prod_{j=1}^{|X'|}(t-x_j)}\,.^{3}$$

Let $X(z), X'(z)$ be their characteristic sets and $Y(z) = X(z) - X'(z)$. Observe that $Y(t) = 0$ hence t is a root of $Y(z)$. By running a randomized polynomial factoring algorithm on $Y(z)$, an adversary can find t with non-negligible probability, thus breaking q-SDH.

By a simple union bound, the two first events can only happen with negligible probability. Conditioned on that not happening, the proof π is an accepting proof for a SNARK for the language $\mathcal{L}(\mathcal{C})$ for input sets $X_{i_j}, \ldots, X_{i_{N-1}}$ and output y. A cheating adversary can extract these sets with overwhelming probability from q-PKE, hence it can be used to break the security of our SNARK for sets. Since this can happen with negligible probability from Theorem 2, the security claim follows.

[3] In fact, the characteristic polynomial of X' will be extracted. The same approach we explain here can be used to break the q-SDH assumption even if the extracted polynomial is not the characteristic polynomial of some set.

The complexities follow trivially from the complexities of the related algorithms of the SNARK except for $\mathsf{Compute}$ where $N-1$ Merkle proofs must be computed, each in time $\log m$. ■

While the above construction yields a construction that appears practical, it has the downside that it produces proofs that are linear in the number of input sets $O(N)$ in size instead of the constant size proofs of our SNARK for sets. One way to circumvent this would be to substitute the part or our scheme that deals with validating that the correct sets were used (the Merkle tree part of the scheme and the λ values in the proof) with an appropriate SNARK that would prove the same statement. One good candidate SNARK would be the construction from [13] for arithmetic circuits. Indeed an arithmetic circuit proving the validity of values a_j would be of size $O(m)$ (assuming a hash computation can be made with a constant number of arithmetic gates). Since both SNARK's in such a construction offer extractability, it would be easy for a cheating adversary to be reduced to an adversary for one of them and this scheme would offer constant size proofs, assuming each of the two SNARKs does so. However, it is not clear how practical such a construction would be and since N is only a the number of input sets (as opposed to, say, their cardinalities), we only present it here as a viable theoretical construction.

6.7 Tight Complexity Analysis for Proof Construction.

Here we give an example of how we can deduce the exact cost related to the proof computation of our SNARK. We will measure the number of necessary exponentiations for the mini-circuit of a single intersection gate with input sets A, B of total cardinality $|\mathsf{A}| + |\mathsf{B}| = D$. Let $\mathsf{I}(z), \alpha(z)$, $\beta(z)$, $\gamma(z)$ and $\delta(z)$ be polynomials as defined in Lemma 2. By the intersection constraints in the lemma, it follows that $deg(\gamma(z)) + deg(\mathsf{I}(z)) = |\mathsf{A}|$ and $deg(\delta(z)) + deg(\mathsf{I}(z)) = |\mathsf{B}|$. Also, by the extended Euclidean algorithm, $deg(\alpha(z)) + \deg(\beta(z)) \leq D$. Therefore the sum of the degrees of all the polynomials at the wires (except for the input wires) is $\leq 2D$.

These polynomials will be used for the computation of seven of the proof elements, namely the six *extractability* terms and the one *consistency* term as defined in Section 3.1, hence the total number of exponentiations for these terms is at most $7 \cdot 2D$. Finally, by the analysis in Section 3.3, polynomial $h(x, z)$ has degree d to variable x and $2D$ in z (since the maximum cardinality of a wire value is D) thus the maximum number of necessary exponentiations to compute $g^{h(s,t)}$ is $2Dd$. This circuit has four multiplication gates therefore $d = 4$ hence the overall necessary exponentiations for the proof computation are at most $14D + 8D = 22D$.