Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation

Berry Schoenmakers[1], Meilof Veeningen[2], and Niels de Vreede[1]

[1] Dept Math & CS, TU Eindhoven, The Netherlands
[2] Philips Research, Eindhoven, The Netherlands

Abstract. Verifiable computation allows a client to outsource computations to a worker with a cryptographic proof of correctness of the result that can be verified faster than performing the computation. Recently, the Pinocchio system achieved faster verification than computation in practice for the first time. Unfortunately, Pinocchio and other efficient verifiable computation systems require the client to disclose the inputs to the worker, which is undesirable for sensitive inputs. To solve this problem, we propose Trinocchio: a system that distributes Pinocchio to three (or more) workers, that each individually do not learn which inputs they are computing on. We fully exploit the almost linear structure of Pinochhio proofs, letting each worker essentially perform the work for a single Pinocchio proof; verification by the client remains the same. Moreover, we extend Trinocchio to enable joint computation with multiple mutually distrusting inputters and outputters and still very fast verification. We show the feasibility of our approach by analysing the performance of an implementation in a case study.

This is the full version (with security proof) of the ACNS'16 paper [31]

Keywords: verifiable computation, quadratic arithmetic programs, privacy, multiparty computation

1 Introduction

Recent cryptographic advances are starting to make verifiable computation more and more practical. The goal of verifiable computation is to allow a client to outsource a computation to a worker and cryptographically verify the result with less effort than performing the computation itself. Based on recent ground-breaking ideas [22, 18], Pinocchio [27] was the first implemented system to achieve this for some realistic computations. Recent works have improved the state-of-the-art in verifiable computation, e.g., by considering better ways to specify computations [5], or adding access control [1].

However, one feature not yet available in practical verifiable computation is privacy, meaning that the worker should not learn the inputs that it is computing on. This feature would enable a client to save time by outsourcing computations, even if the inputs of those computations are so sensitive that it does not want to disclose them to the worker. Also, it would allow verifiable computation to be used in settings where multiple clients do not trust the worker or each other, but still want to perform a joint computation over their respective inputs and be sure of the correctness of the result.

While privacy was already defined in the first paper to formalize verifiable computation [17], it has not been shown so far how it is efficiently achieved, with existing constructions relying on efficient cryptographic primitives. By outsourcing a computation to multiple workers, it *is* possible to guarantee privacy (if not all workers are corrupted) and correctness, but existing constructions from the multiparty literature lose the most appealing feature of verifiable computation: namely, that computations can be verified very quickly, even in time independent from the computation size. This leads to the central question of this paper: can we perform verifiable computation with the *correctness* and *performance* guarantees of [27], but while also getting *privacy* against corrupted workers?

1.1 Our Contributions

In this paper, we introduce Trinocchio to show that indeed, it is possible to outsource a computation in a privacy-preserving way to multiple workers, while retaining the fast verification offered by verifiable computation. Trinocchio uses state-of-the-art [27]-style proofs, but distributes the computation of these proofs to, e.g., three workers such that no single worker learns anything about the inputs. The client essentially gets a normal Pinocchio proof, so we keep Pinocchio's correctness guarantees and fast verification. The critical observation is that the almost linear structure of Pinocchio proofs (supporting verification based on bilinear maps) allows us to distribute the computation of Pinocchio proofs such that individual workers perform essentially the same work as a normal Pinocchio prover in the non-distributed setting. Specifically, our contributions are:

- We show how to distribute the production of Pinocchio proofs in a privacy-preserving way to multiple workers, thereby achieving privacy-preserving verifiable computation in the setting with one client
- We extend our system to settings with multiple distrusting input and result parties
- We provide a precise security model capturing the security guarantees of our protocols: privacy, correctness, but also input independence
- We demonstrate the practical feasibility of our approach by implementing a case study: we demonstrate Trinocchio's low overhead by repeating the multivariate polynomial evaluation case study of [27]'s.

While our Trinocchio protocol ensures correct function evaluation, it only fully protects privacy against semi-honest workers. This is a realistic attacker model; in particular, it means that side channel attacks on individual workers are ineffective because each individual worker's communication and computation are completely independent from the sensitive inputs. However, even if an adversary should be able to obtain sensitive information, they are unable to manipulate the result thanks to the use of verifiable computation. In this way, our protocol *hedges* against the risk of more powerful adversaries.

1.2 Related Work

Privacy-preserving outsourcing to single workers has been considered in the literature, but constructions in this setting rely on inefficient cryptographic primitives like fully homomorphic encryption [17, 11, 16], functional encryption [20], and multi-input attribute-based encryption [21]. (This is not surprising: indeed, even without guaranteeing correctness, letting a single worker perform a computation on inputs it does not know would intuitively seem to require some form of fully homomorphic encryption.) Some of these works also consider a multi-client setting [11, 21].

A large body of works considers multiparty computation for privacy-preserving outsourcing (see, e.g., [24, 28, 9, 23]). These works do not consider verifiability and achieve correctness at best in the case that *all-but-one* workers are corrupt (due to inherent limitations of the underlying protocols). We stress that this is rather unsatisfactory for the outsourcing scenario, where one naturally wishes to cover the case that *all* workers are corrupt—dispensing of the need to trust any particular worker.

Concerning outsourcing to multiple workers, [2] presents a verifiable computation protocol combining privacy and correctness; but unfortunately, they guarantee neither privacy nor correctness if all workers are corrupted and may collude; and it places a much higher burden on the workers than, e.g., [27]. Alternatively, recent works [3, **?**,30], like us, guarantee correctness independent of worker corruption, but privacy only under some conditions. Our work offers a substantial performance improvement over these works by fully exploiting a set-up that needs to be trusted both for guaranteeing privacy and for guaranteeing correctness.

We should mention that the notion of verifiability exists in various forms and the field has a richer background than presented here, however, we focus entirely on the notion of verifiable computation first formalized by [17], because it is tailored to the outsourcing scenario.

1.3 Outline

We first briefly define the security model for privacy-preserving outsourced computation in Section 2. In Section 3, we show how Trinocchio distributes the proof computation of Pinocchio in the single-client scenario, and prove security of the construction. We generalise Trinocchio to the setting with multiple, mutually distrusting inputters and outputters in Section 4. Finally, we demonstrate the feasibility of Trinocchio in Section 5 by analysing its performance in a case study, computing a multivariate polynomial evaluation. We finish with a discussion and conclusions in Section 6.

For convenience, we also provide a brief overview of the Pinocchio protocol [27] for verifiable computation based on quadratic arithmetic programs in Appendix A.

Secure function evaluation:

- Honest parties send inputs x_i to trusted party
- Adversary sends inputs x_i of corrupted parties to trusted party (active adversary may modify them)
- Trusted party computes function $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ (where $y_1 = \ldots = \perp$ if any $x_i = \perp$)
- Trusted party provides outputs y_i for corrupted parties to adversary
- Trusted party provides outputs y_i to honest parties
- Honest parties output received value; corrupted parties output $\perp$; adversary chooses own output

Correct function evaluation:

- Honest parties send inputs x_i to trusted party
- Adversary sends inputs x_i of corrupted parties to trusted party (active adversary may modify them)
- Trusted party computes function $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ (where $y_1 = \ldots = \perp$ if any $x_i = \perp$)
- Trusted party provides all inputs x_i to adversary
- Adversary gives subset of honest parties to trusted party (passive adversary gives all honest parties)
- Trusted party sends outputs y_i to given honest parties, $\perp$ to others
- Honest parties output received value; corrupted parties output $\perp$; adversary chooses own output

Fig. 1. Ideal-world executions of secure (left) and correct (right) function evaluation. The highlighted text indicates where the two differ.

2 Security Model for Privacy-Preserving Outsourcing

In this section, we define security for privacy-preserving outsourcing. Because we have interactive protocols between multiple parties (as opposed to a cryptographic scheme, like verifiable computation above), we define security using the ideal/real-paradigm [6]. In our setting, the parties are several *result parties* that wish to obtain the result of a computation on inputs held by several *input parties*, who are willing to enable the computation, but not to divulge their private input values to anybody else. Therefore, they outsource the computation to several *workers*. (Input and result parties may overlap.) The simplest case is the "single-client scenario" in which one party is the single input/result party.

We consider protocols operating in three phases: an *input phase* involving the input parties and workers; a *computation phase* involving only the workers; and a *result phase* involving the workers and result parties. The work of the input parties and output parties should depend only on the number of other parties and the size of their own in/outputs.

To define security, we will re-use the existing definition framework for secure function evaluation [6]. These definitions not specific to the outsourcing setting; but the outsourcing setting will become apparent when we claim that a protocol, e.g., implements secure function evaluation *if at most X workers are corrupted*. Secure function evaluation is the problem to evaluate $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ with m parties such that the ith party inputs x_i and obtains y_i, and no party learns anything else. (In outsourcing, result parties have non-empty output, input parties have non-empty inputs, and workers have empty in- and outputs.) A protocol π *securely evaluates function* f if the outputs of the parties and adversary $\mathcal{A}$ in a real-world execution of the protocol can be emulated by the outputs of the parties and an adversary $\mathcal{S}_{\mathcal{A}}$ in an idealised execution, where f is computed by a trusted party that acts as shown in Figure 1. Security is guaranteed because the trusted party correctly computes the function. Privacy is guaranteed because the adversary in the idealised execution does not learn anything it should not. Secure evaluation also implies *input independence*, meaning that an input party cannot let its input depend on that of another, e.g., by copying the input of another party; this is guaranteed because the adversary needs to provide the inputs of corrupted parties without seeing the honest inputs. Typically, protocols achieve secure function evaluation for a given, restricted class of adversaries, e.g., adversaries that are passive and only corrupt a certain number of workers. Protocols can require set-up assumptions; these are captured by giving protocol participants access to a set of functions $g_1, \ldots, g_k$ that are always evaluated correctly. In this case, we say that the protocol securely evaluates the function *in the* $(g_1, \ldots, g_k)$*-hybrid model.* For details, see [6].

We only achieve secure function evaluation if not too many workers are corrupted; we still need to formalise that in all other cases, we still guarantee that the function was evaluated correctly. This weaker security guarantee, which we call *correct function evaluation*, captures security and input independence, as above, but not privacy. It is formalised by modifying the ideal-world execution as shown in Figure 1. Namely, after evaluating f, the trusted party provides all inputs to the adversary (modelling that the computation may leak the inputs), who, based on these inputs, can decide which honest parties are allowed to see their outputs. Hence, we guarantee that, *if* an honest party gets a result, then it gets the correct result of the computation on independently chosen inputs, but not that the inputs remain hidden, or that it gets a result at all. Note that, in this definition, the adversary has complete control over which result parties see an output and which ones do not.

3 Distributing the Prover Computation

In this section, we present the single-client version of our Trinocchio protocol for privacy-preserving outsourcing. In Trinocchio, a client distributes computation of a function $x_2 = f(x_1)$ to n workers (we consider here single-valued input and output, but the generalisation is straightforward). Trinocchio guarantees correct function evaluation (regardless of corruptions) and secure function evaluation (if at most θ workers are passively corrupted, where $n = 2\theta + 1$). Trinocchio in effect distributes the proof computation of Pinocchio; the number of workers to obtain privacy against one semi-honest worker is three, hence its name.

3.1 Multiparty Computation using Shamir Secret Sharing

To distribute the Pinocchio computation, Trinocchio employs multiparty computation techniques based on Shamir secret sharing [4]. Recall that in (θ, n)

Shamir secret sharing, a party shares a secret s among n parties so that $\theta + 1$ parties are needed to reconstruct s. It does this by taking a random degree-$\leq \theta$ polynomial $p(x) = \alpha_\theta x^\theta + \ldots + \alpha x + s$ with s as constant term and giving $p(i)$ to party i. Since $p(x)$ is of degree at most θ, $p(0)$ is completely independent from any θ shares but can be easily computed from any $\theta + 1$ shares by Lagrange interpolation. We denote such a sharing as $[\![s]\!]$. Note that Shamir-sharing can also be done "in the exponent", e.g., $[\![\langle a \rangle_1]\!]$ denotes a Shamir sharing of $\langle a \rangle_1 \in \mathbb{G}_1$ from which $\langle a \rangle_1$ can be computed using Lagrange interpolation in $\mathbb{G}_1$.

Shamir secret sharing is linear, i.e., $[\![a + b]\!] = [\![a]\!] + [\![b]\!]$ and $[\![\alpha a]\!] = \alpha [\![a]\!]$ can be computed locally. When computing the product of $[\![a]\!]$ and $[\![b]\!]$, each party i can locally multiply its points $p_a(i)$ and $p_b(i)$ on the random polynomials p_a and p_b. Because the product polynomial has degree at most 2θ, this is a $(2\theta, n)$ sharing, which we write as $[a \cdot b]$ (note that reconstructing the secret requires $n = 2\theta + 1$ parties). Moreover, the distribution of the shares of $[a \cdot b]$ is not independent from the values of a and b, so when revealed, these shares reveal information about a and b. Hence, in multiparty computation, $[a \cdot b]$ is typically converted back into a random (θ, n) sharing $[\![a \cdot b]\!]$ using an interactive protocol due to [19]. Interactive protocols for many other tasks such as comparing two shared value also exist (see, e.g., [15]).

3.2 The Trinocchio protocol

We now present the Trinocchio protocol. Trinocchio assumes that Pinocchio's KeyGen has been correctly performed: formally, Trinocchio works in the KeyGen-hybrid model. Furthermore, Trinocchio assumes pairwise private, synchronous communication channels. To obtain $x_2 = f(x_1)$, a client proceeds in four steps:

- The client obtains the verification key, and the workers obtain the evaluation key, using hybrid calls to KeyGen.
- The client secret shares $[\![x_1]\!]$ of its input to the workers.
- The workers use multiparty computation to compute secret-shares $[\![x_2]\!]$ of the output and $[\![\langle V_{\mathrm{mid}} \rangle_1]\!]$, $[\![\langle \alpha_v V_{\mathrm{mid}} \rangle_1]\!]$, $[\![\langle W_{\mathrm{mid}} \rangle_2]\!]$, $[\![\langle \alpha_w W_{\mathrm{mid}} \rangle_1]\!]$, $[\![\langle Y_{\mathrm{mid}} \rangle_1]\!]$, $[\![\langle \alpha_y Y_{\mathrm{mid}} \rangle_1]\!]$, $[\![\langle Z \rangle_1]\!]$, $[\langle H \rangle_1]$ of the Pinocchio proof, as we explain next; and sends these shares to the client.
- The client recombines the shares into $\langle V_{\mathrm{mid}} \rangle_1$, $\langle \alpha_v V_{\mathrm{mid}} \rangle_1$, $\langle W_{\mathrm{mid}} \rangle_2$, $\langle \alpha_w W_{\mathrm{mid}} \rangle_1$, $\langle Y_{\mathrm{mid}} \rangle_1$, $\langle \alpha_y Y_{\mathrm{mid}} \rangle_1$, $\langle Z \rangle_1$, $\langle H \rangle_1$ by Lagrange interpolation, and accepts x_2 as computation result if Pinocchio's Verify returns success.

Algorithm 1 shows in detail how the secret-shares of the function output and Pinocchio proof are computed. The first step is to compute function output $x_2 = f(x_1)$ and values $(x_3, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of the QAP (line 4). This is done using normal multiparty computation protocols based on secret sharing. If function f is represented by an arithmetic circuit, then it is evaluated using local addition and scalar multiplication, and the multiplication protocol from [19]. If f is represented by a circuit using more complicated gates, then specific protocols may be used: e.g., the split gate discussed in Appendix A.1 can be evaluated using multiparty bit decomposition protocols [13,

Algorithm 1 Trinocchio's $\mathsf{Compute}$ protocol

```
1: ▷ $\mathcal{S} = \{\alpha_1, \ldots, \alpha_d\}$ denotes the list of roots of the target polynomial of the QAP
2: ▷ $\mathcal{T} = \{\beta_1, \ldots, \beta_d\}$ denotes a list of distinct points different from $\mathcal{S}$
3: function Compute($\mathsf{EK}_f = \{\langle r_v v_i \rangle_1\}_i, \ldots, \{\langle s^j \rangle_1\}_j; [\![x_1]\!]$)
4:     $([\![x_2]\!], \ldots, [\![x_k]\!]) \leftarrow f([\![x_1]\!])$
5:     $[\![\boldsymbol{v}]\!] \leftarrow \{\sum_i v_i(\alpha_j) \cdot [\![x_i]\!]\}_j$; $[\![\boldsymbol{V}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{v}]\!])$; $[\![\boldsymbol{v}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{V}]\!])$
6:     $[\![\boldsymbol{w}]\!] \leftarrow \{\sum_i w_i(\alpha_j) \cdot [\![x_i]\!]\}_j$; $[\![\boldsymbol{W}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{w}]\!])$; $[\![\boldsymbol{w}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{W}]\!])$
7:     $[\![\boldsymbol{y}]\!] \leftarrow \{\sum_i y_i(\alpha_j) \cdot [\![x_i]\!]\}_j$; $[\![\boldsymbol{Y}]\!] \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}([\![\boldsymbol{y}]\!])$; $[\![\boldsymbol{y}']\!] \leftarrow \mathsf{FFT}_{\mathcal{T}}([\![\boldsymbol{Y}]\!])$
8:     $[\boldsymbol{h}'] \leftarrow \{([\![\boldsymbol{v}'_j]\!] \cdot [\![\boldsymbol{w}'_j]\!] - [\![\boldsymbol{y}'_j]\!])/t(\beta_j)\}_j$; $[\boldsymbol{H}] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\boldsymbol{h}'])$
9:     $[\![\langle V_{\mathrm{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_v v_i \rangle_1 \cdot [\![x_i]\!]$
10:    $[\![\langle \alpha_v V_{\mathrm{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot [\![x_i]\!]$
11:    $[\![\langle W_{\mathrm{mid}} \rangle_2]\!] \leftarrow \sum_i \langle r_w w_i \rangle_2 \cdot [\![x_i]\!]$
12:    $[\![\langle \alpha_w W_{\mathrm{mid}} \rangle_1]\!] \sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot [\![x_i]\!]$
13:    $[\![\langle Y_{\mathrm{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_y y_i \rangle_1 \cdot [\![x_i]\!]$
14:    $[\![\langle \alpha_y Y_{\mathrm{mid}} \rangle_1]\!] \leftarrow \sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot [\![x_i]\!]$
15:    $[\![\langle Z \rangle_1]\!] \leftarrow \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot [\![x_i]\!]$
16:    $[\langle H \rangle_1] = \sum_j \langle s^j \rangle_1 \cdot [\boldsymbol{H}_j]$
17:    return ($[\![x_2]\!]; [\![\langle V_{\mathrm{mid}} \rangle_1]\!], [\![\langle \alpha_v V_{\mathrm{mid}} \rangle_1]\!], [\![\langle W_{\mathrm{mid}} \rangle_2]\!], [\![\langle \alpha_w W_{\mathrm{mid}} \rangle_1]\!]$,
18:                $[\![\langle Y_{\mathrm{mid}} \rangle_1]\!], [\![\langle \alpha_y Y_{\mathrm{mid}} \rangle_1]\!], [\![\langle Z \rangle_1]\!], [\langle H \rangle_1]$)
```

29]. Any protocol can be used as long as it guarantees privacy, i.e., the view of any θ workers is statistically independent from the values represented by the shares.

The next task is to compute, in secret-shared form, the coefficients of the polynomial $h = ((\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i))/t \in \mathbb{F}[x]$ that we need for proof element $\langle H \rangle_1$. In theory, this computation could be performed by first computing shares of the coefficients of $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, and then dividing by t, which can be done locally using traditional polynomial long division. However, this scales quadratically in the degree of the QAP and hence leads to unacceptable performance. Hence, we take the approach based on fast Fourier transforms (FFTs) from [5], and adapt it to the distributed setting. Given a list $\mathcal{S} = \{\omega_1, \ldots, \omega_d\}$ of distinct points in $\mathbb{F}$, we denote by $\boldsymbol{P} = \mathsf{FFT}_{\mathcal{S}}(\boldsymbol{p})$ the transformation from coefficients $\boldsymbol{p}$ of a polynomial p of degree at most $d-1$ to evaluations $p(\omega_1), \ldots, p(\omega_d)$ in the points in $\mathcal{S}$. We denote by $\boldsymbol{p} = \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{P})$ the inverse transformation, i.e., from evaluations to coefficients. Deferring specifics to later, we mention now that the FFT is a linear transformation that, for some $\mathcal{S}$, can be performed locally on secret shares in $\mathcal{O}(d \cdot \log d)$.

With FFTs available, we can compute the coefficients of h by evaluating h in d distinct points and applying FFT^{-1}. Note that we can efficiently compute evaluations $\boldsymbol{v}$ of $v = (\sum_i x_i v_i)$, $\boldsymbol{w}$ of $w = (\sum_i x_i w_i)$, and $\boldsymbol{y}$ of $y = (\sum_i x_i y_i)$ in the zeros $\{\omega_1, \ldots, \omega_d\}$ of the target polynomial. Namely, the values $v_k(\omega_i)$, $w_k(\omega_i)$, $y_k(\omega_i)$ are simply the coefficients of the quadratic equations represented by the QAP, most of which are zero, so these sums have much fewer than k elements (if this were not the case, then evaluating v, w, and y would take an unacceptable $O(d \cdot k)$). Unfortunately, we cannot use these evaluations directly to obtain evaluations of h, because this requires division by the target polynomial, which is zero in exactly these points ω_i. Hence, after determining $\boldsymbol{v}$, $\boldsymbol{w}$, and $\boldsymbol{y}$, we first use the inverse FFT to determine the coefficients $\boldsymbol{V}$, $\boldsymbol{W}$, and $\boldsymbol{Y}$ of v, w, and y, and then again the FFT to compute the evaluations $\boldsymbol{v'}$, $\boldsymbol{w'}$, and $\boldsymbol{y'}$ of v, w, and y in another set of points $\mathcal{T} = \{\Omega_1, \dots, \Omega_k\}$ (lines 5–7). Now, we can compute evaluations $\boldsymbol{h'}$ of h in $\mathcal{T}$ using $h(\Omega_i) = (v(\Omega_i) \cdot w(\Omega_i) - y(\Omega_i))/t(\Omega_i)$. This requires a multiplication of (θ, n)-secret shares of $v(\Omega_i)$ and $w(\Omega_i)$, hence the result is a $(2\theta, n)$-sharing. Finally, the inverse FFT gives us a $(2\theta, n)$-sharing of the coefficients $\boldsymbol{H}$ of h (line 8).

Given secret shares of the values of x_i and coefficients of h, it is straightforward to compute secret shares of the Pinocchio proof. Indeed, $\langle V_{\text{mid}} \rangle_1, \dots, \langle H \rangle_1$ are all computed as linear combinations of elements in the evaluation key, so shares of these proof elements can be computed locally (lines 9–16), and finally returned by the respective workers (lines 17–18).

Note that, compared to Pinocchio, our client needs to carry out slightly more work. Namely, our client needs to produce secret shares of the inputs and recombine secret shares of the outputs; and it needs to recombine the Pinocchio proof. However, according to the micro-benchmarks from [27], this overhead is small. For each input and output, Verify includes three exponentiations, whereas $\mathsf{Combine}$ involves four additions and two multiplications; when using [27]'s techniques, this adds at most a 3% overhead. Recombining the Pinocchio proof involves 15 exponentiations at around half the cost of a single pairing. Alternatively, it is possible to let one of the workers perform the Pinocchio recombining step by using the distributed zero-knowledge variant of Pinocchio (Appendix A.3) and the techniques from Section 4. In this case, the only overhead for the client is the secret-sharing of the inputs and zero-knowledge randomness, and recombining the outputs.

Parameters for Efficient FFTs To obtain efficient FFTs, we use the approach of [5]. There, it is noted that the operation $\boldsymbol{P} = \mathsf{FFT}_{\mathcal{S}}(\boldsymbol{p})$ and its inverse can be efficiently implemented if $\mathcal{S} = \{\omega, \omega^2, \dots, \omega^d = 1\}$ is a set of powers of a primitive dth root of unity, where d is a power of two. (We can always demand that QAPs have degree $d = 2^k$ for some k by adding dummy equations.) Moreover, [5] presents a pair of groups $\mathbb{G}_1, \mathbb{G}_2$ of order q such that $\mathbb{F}_q$ has a primitive 2^{30}th root of unity (and hence also primitive 2^kth roots of unity for any $k < 30$) as well as an efficiently computable pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$. Finally, [5] remarks that for $\mathcal{T} = \{\eta\omega, \eta\omega^2, \dots, \eta\omega^d = \eta\}$, operations $\mathsf{FFT}_{\mathcal{T}}^{-1}$ and $\mathsf{FFT}_{\mathcal{T}}^{-1}$ can easily be reduced to $\mathsf{FFT}_{\mathcal{S}}$ and $\mathsf{FFT}_{\mathcal{S}}^{-1}$, respectively. In our implementation, we use exactly these suggested parameters.

3.3 Security of Trinocchio

Theorem 1. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the Trinocchio protocol. Assuming the d-PKE, $(4d+4)$-PDH, and $(8d+8)$-SDH assumptions:*

– *Trinocchio correctly evaluates f in the* KeyGen*-hybrid model.*

– *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the* KeyGen*-hybrid model.*

The proof of this theorem is easily derived as a special case of the proof for the multi-client Trinocchio protocol later. Here, we present a short sketch.

Proof (Sketch). To prove correct function evaluation, we need to show that for every real-world adversary $\mathcal{A}$ interacting with Trinocchio, there is an ideal-world simulator $\mathcal{S}_{\mathcal{A}}$ that interacts with the trusted party for correct function evaluation such that the two executions give indistinguishable results. The only interesting case is when the client is honest and some of the workers are not. In this case, the simulator receives the input of the honest party, and needs to choose whether to provide the output. To this end, the simulator simply simulates a run of the actual protocol with $\mathcal{A}$, until it has finally obtained function output x_2 and the accompanying Trinocchio proof. If the proof verifies, it tells the trusted party to provide the output to the client; otherwise, it tells the trusted party not to. Finally, the simulator outputs whatever $\mathcal{A}$ outputs. Because Trinocchio is secure, except with negligible probability a verifying proof implies that the real-world output of the client (as given by the adversary) matches the ideal-world output of the client (as computed by the trusted party); and by construction, the outputs of $\mathcal{A}$ and $\mathcal{S}_{\mathcal{A}}$ are distributed identically. This proves correct function evaluation.

For secure function evaluation, again the only interesting case is if the client is honest and some of the workers are passively corrupted. In this case, because corruption is only passive, correctness of the multiparty protocol used to compute f and correctness of the Pinocchio proof system used to compute the proof together imply that real-world executions (like ideal-world executions) result in the correct function result and a verifying proof. Hence, we only need to worry about how $\mathcal{S}_{\mathcal{A}}$ can simulate the view of $\mathcal{A}$ on the Trinocchio protocol without knowing the client's input. However, note that the workers only use a multiparty computation to compute f (which we assume can be simulated without knowing the inputs), after which they no longer receive any messages. Hence simulating the multiparty computation for f and receiving any messages that $\mathcal{A}$ sends is sufficient to simulate $\mathcal{A}$. This proves secure function evaluation. □

Privacy against Active Attacks We remark that actually, Trinocchio in some cases provides privacy against corrupted workers as well. Namely, suppose that the protocol used to compute f does not leak any information to corrupted workers in the event of an active attack (even though in this case it may not guarantee correctness). For instance, this is the case for the protocol from [19]: the attacker can manipulate the shares that it sends, which makes the computation return incorrect results; but since the attacker always learns only θ many shares of any value, it does not learn any information. Because the attacker learns no additional information from producing the Pinocchio proof, the overall protocol still leaks no information to the adversary. (And security of Pinocchio ensures the client notices the attacker's manipulation.)

This crucially relies on the workers not learning whether the client accepts the proof: if the workers would learn whether the client obtained a validating proof,

Algorithm 2 ProofBlock

```
1: function ProofBlock($BK; \boldsymbol{x}; \delta_v, \delta_w, \delta_y$)
2:     $\langle V\rangle_1 \leftarrow \langle r_v t\rangle_1 \delta_v + \sum_i \langle r_v v_i\rangle_1 x_i$; $\langle V'\rangle_1 \leftarrow \langle r_v \alpha_v t\rangle_1 \delta_v + \sum_i \langle r_v \alpha_v v_i\rangle_1 x_i$
3:     $\langle W\rangle_2 \leftarrow \langle r_w t\rangle_2 \delta_w + \sum_i \langle r_w w_i\rangle_2 x_i$; $\langle W'\rangle_1 \leftarrow \langle r_w \alpha_w t\rangle_1 \delta_w + \sum_i \langle r_w \alpha_w w_i\rangle_1 x_i$
4:     $\langle Y\rangle_1 \leftarrow \langle r_y t\rangle_1 \delta_y + \sum_i \langle r_y y_i\rangle_1 x_i$; $\langle Y'\rangle_1 \leftarrow \langle r_y \alpha_y t\rangle_1 \delta_y + \sum_i \langle r_y \alpha_y y_i\rangle_1 x_i$
5:     $\langle Z\rangle_1 \leftarrow \langle r_v \beta t\rangle_1 \delta_v + \langle r_w \beta t\rangle_1 \delta_w + \langle r_y \beta t\rangle_1 \delta_y + \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i\rangle_1 x_j$
6:     return $(\langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1)$
```

then, by manipulating proof construction, they could learn whether a modified version of the tuple $(x_1, \ldots, x_k)$ is a solution of the QAP used, so corrupted workers could learn one chosen bit of information about the inputs (cf. [26]).

4 Handling Mutually Distrusting In- and Outputters

We now consider the scenario where there are multiple (possibly overlapping) input and result parties. There are some significant changes between this scenario and the single-client scenario. In particular, we need to extend Pinocchio to allow verification not based on the actual input/output values (indeed, no party sees all of them) but on some kind of representation that does not reveal them. Moreover, we need to use the zero-knowledge variant of Pinocchio (Appendix A.3), and we need to make sure that input parties choose their inputs independently from each other.

4.1 Multi-Client Proofs and Keys

Our multi-client Trinocchio proofs are a generalisation of the zero-knowledge variant of Pinocchio (Appendix A.3) with modified evaluation and verification keys. Recall that in Pinocchio, the proof terms $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, and $\langle Z\rangle_1$ encode circuit values $x_{l+m+1}, \ldots, x_k$; in the zero-knowledge variant, these terms are randomised so that they do not reveal any information about $x_{l+m+1}, \ldots, x_k$. In the multi-client case, additionally, the inputs of all input parties and the outputs of all result parties need to be encoded such that no other party learns any information about them. Therefore, we extend the proof with *blocks* of the above seven terms for each input and result party, which are constructed in the same way as the seven proof terms above. Although some result parties could share a block of output values, for simplicity we assign each result party its own block in the protocol.

To produce a block containing values $\boldsymbol{x}$, a party first samples three random field values δ_v, δ_w, and δ_y and then executes ProofBlock, cf. Algorithm 2. The BK argument to this algorithm is the *block key*; the subset of the evaluation key terms specific to a single proof block. Because each input party should only provide its own input values and should not affect the values contributed by other parties, each proof block must be restricted to a subset of the wires. This is achieved by modifying Pinocchio's key generation such that, instead of a sampling a single

Algorithm 3 CheckBlock

```
1: function CheckBlock($BV; \langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1$)
2:     if $e(\langle V\rangle_1, \langle\alpha_v\rangle_2) = e(\langle V'\rangle_1, \langle 1\rangle_2)$
3:      $\wedge e(\langle\alpha_w\rangle_1, \langle W\rangle_2) = e(\langle W'\rangle_1, \langle 1\rangle_2)$
4:      $\wedge e(\langle Y\rangle_1, \langle\alpha_y\rangle_2) = e(\langle Y'\rangle_1, \langle 1\rangle_2)$
5:      $\wedge e(\langle Z\rangle_1, \langle 1\rangle_2) = e(\langle V\rangle_1 + \langle Y\rangle_1, \langle\beta\rangle_2)e(\langle\beta\rangle_1, \langle W\rangle_2)$ then
6:         return $\top$
7:     else
8:         return $\bot$
```

value β, one such value, β_j, is sampled for each proof block j and the terms $\langle r_v\beta_j v_i + r_w\beta_j w_i + r_y\beta_j y_i\rangle_1$ are only included for wires indices i belonging to block j. That is, the jth block key is

$$BK_j = \{\langle r_v v_i\rangle_1, \langle r_v\alpha_v v_i\rangle_1, \langle r_w w_i\rangle_2, \langle r_w\alpha_w w_i\rangle_1, \langle r_y y_i\rangle_1, \langle r_y\alpha_y y_i\rangle_1,$$
$$\langle r_v\beta_j v_i + r_w\beta_j w_i + r_y\beta_j y_i\rangle_1, \langle r_v\beta_j t\rangle_1, \langle r_w\beta_j t\rangle_1, \langle r_y\beta_j t\rangle_1\},$$

with i ranging over the indices of wires in the block. Note that ProofBlock only performs linear operations on its $\boldsymbol{x}$, δ_v, δ_w and δ_y inputs. Therefore this algorithm does not have to be modified to compute on secret shares.

A Trinocchio proof in the multi-client setting now consists of one block $\boldsymbol{Q}_i = (\langle V_i\rangle_1, \ldots, \langle Z_i\rangle_1)$ for each input and result party, one block $\boldsymbol{Q}_{\text{mid}} = (\langle V_{\text{mid}}\rangle_1, \ldots, \langle Z_{\text{mid}}\rangle_1)$ of internal wire values, and Pinocchio's $\langle H\rangle_1$ element. Verification of such a proof consists of checking correctness of each block, and checking correctness of $\langle H\rangle_1$. The validity of a proof block can be verified using CheckBlock, cf. Algorithm 3. Compared to the Pinocchio verification key, our verification key contains "block verification keys" BV_i (i.e., elements $\langle\beta_j\rangle_1$ and $\langle\beta_j\rangle_2$) for each block instead of just $\langle\beta\rangle_1$ and $\langle\beta\rangle_2$. Apart from the relations inspected by CheckBlock, one other relation is needed to verify a Pinocchio proof: the divisibility check of Equation (4) (Appendix A.2). In the protocol, the algorithm that verifies this relation will be called CheckDiv. We denote the modified setup of the evaluation and verification keys by hybrid call MKeyGen.

4.2 Protocol Overview

We will proceed with a protocol overview. Pseudocode and a more detailed description of the protocol are given in Appendix B. The multi-client variant of our Trinicchio protocol makes use of private channels, just as the single-client variant, to privately communicate in- and output values, and to let the workers carry out the computation. We need some additional communication to ensure input independence and fix the input parties' values. For this we use a bulletin board. To achieve input independence, we first have the input parties commit to a representation of their input and then reveal these, which requires the use of a commitment scheme.

Apart from key set-up there are three phases to the multi-client Trinocchio protocol.

- In the *input phase*, the input parties provide representations of their input on the bulletin board. These representations are later used as part of the proof to verify the computation results. They also serve to ensure that each input party provides its value independent of the other input values. The input parties then secret share their input values to the workers. The workers verify that the secret shared input values are consistent with their representations on the bulletin board, to prevent malicious input parties from providing a different value.
- The *computation phase* is very similar to the single-client variant of Trinocchio. In this phase the workers perform multi-party computation to carry out the actual computation and obtain secret shares of intermediate and result wire values. They then use these secret shared wire values to construct shares of the proof elements. These are then posted on the bulletin board, instead of being communicated directly to the result parties to ensure that all result parties receive a consistent result. In order to prevent these proof elements from revealing any information about the wire values, the zero-knowledge variant of the proof is used (Appendix A.3).
- In the *result phase* the workers privately send the shares of the result values to the result parties. The result parties recombine the proof shares from the bulletin board and check whether the proof verifies. The result parties further check whether the recombined shares of the result are consistent with the information on the bulletin board. The result parties only accept the result received from the workers if both checks are satisfied.

4.3 Security of the Trinocchio Protocol

Analogously to the single-client case, we obtain the following result:

Theorem 2. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the multi-client Trinocchio protocol. Assuming the d-PKE, $(4d + 4)$-PDH, and $(8d + 8)$-SDH assumptions:*

- *Trinocchio correctly evaluates f in the $(\mathsf{ComGen}, \mathsf{MKeyGen})$-hybrid model.*
- *Whenever at most θ workers are passively corrupted, Trinocchio securely evaluates f in the $(\mathsf{ComGen}, \mathsf{MKeyGen})$-hybrid model.*

We stress that "at most θ workers are passively corrupted" includes both the case when the adversary is passively corrupted, and corrupts at most θ workers (as well as arbitrarily many input and result parties); and the case when the adversary is actively corrupted, and corrupts no workers (but arbitrarily many input and result parties)

We give a proof of this theorem in Appendix B. To prove secure function evaluation, we obtain privacy by simulating the multiparty computation of the proof with respect to the adversary without using honest inputs. To prove correct function evaluation, we run the protocol together with the adversary: if this gives a fake Pinocchio proof, then one of the underlying problems can be broken.

In the single-client case, we remarked that Trinocchio actually provides security against up to θ *actively* corrupted workers. Namely, although θ actively corrupted workers may manipulate the computation of the function and proof, they do not learn any information from this because they do not see the resulting proof that the client gets. In our multi-client protocol, it is less natural to assume that the workers cannot see the resulting proof; and in fact, in our protocol, corrupted workers *do* see the full proof as it is posted on the bulletin board. It should be possible to obtain some privacy guarantees against actively malicious workers (who do not collude with any result parties) by letting the result parties provide proof contributions directly to the result parties instead of posting them on the bulletin board. We leave an analysis for future work.

5 Performance

In this section, we show that our approach indeed adds privacy to verifiable computation with little overhead. We demonstrate this in a case study: we take the "MultiVar Poly" application from [27], and show that using Trinocchio, this computation can be outsourced in a private and correct way at essentially the same cost as letting three workers each perform the Pinocchio computation.

In our experiments, one client outsources the computation to three workers. In particular, we use multiparty computation based on $(1, 3)$ Shamir secret sharing. As discussed in Sections 3.3 and 4.3, this guarantees privacy against one passively corrupted worker (or, in the single-client case against θ actively corrupted workers when the multiparty computation protocol does not leak any information). We did not implement the multiple client scenario; this would add small overhead for the workers, with verification effort growing linearly in he number of input and result parties but remaining small and independent from the computation size. To simulate a realistic outsourcing scenario, we distribute computations between three Amazon EC2 "m3.medium" instances[3] around the world: one in Oregon, United States; one in Ireland; and one in Tokyo, Japan. Multiparty computation requires secure and private channels: these are implemented using SSL.

5.1 Case Study: Multivariate Polynomial Evaluation

In [27], Pinocchio performance numbers are presented showing that, for some applications, Pinocchio verification is faster than native execution. One of these applications, "MultiVar Poly", is the evaluation of a constant multivariate polynomial on five inputs of degree 8 ("medium") or 10 ("large"). In this case study, we use Trinocchio to add privacy to this outsourcing scenario.

We have made an implementation[4] of Trinocchio's Compute algorithm (Algorithm 1) that is split into two parts. The first part performs the evaluation of

[3] Running Intel Xeon E5-2670 v2 Ivy Bridge with 4 GB SSD and 3.75 GiB RAM
[4] Implementation available at `http://meilof.home.fmf.nl/`

	# mult	Pinoc.	Dist f	Dist π	Trinoc.	Verif.
MultiVar Poly, Medium	203428	2102	96	2092	2187	0.04
MultiVar Poly, Large	571046	6458	275	6427	6702	0.05

Table 1. Performance of multivariate polynomial evaluation with Trinocchio: number of multiplications in f; time for single-worker proof; time per party for computing f and proof, and total; and verification time (all times in seconds)

the function f (line 4), given as an arithmetic circuit, using the secret sharing implementation of VIFF. (We use the arithmetic circuit produced by the Pinocchio compiler, hence f is exactly the same as in [27].) Note that, because f is an arithmetic circuit, this step does not leak any information against actively corrupted workers. Hence, in the single-client outsourcing scenario of Section 3, we achieve privacy against one actively corrupted worker. The second part is a completely new implementation of the remainder of Trinocchio using [25]'s implementation of the discrete logarithm groups and pairings from [5].

Table 1 shows the performance numbers of running this application in the cloud with Trinocchio. Significantly, evaluating the function f using passively secure multiparty computation (i.e., line 4 of Compute) is more than twenty times cheaper than computing the Pinocchio proof (i.e., lines 5–16 of Comp). Moreover, we see that computing the Pinocchio proof in the distributed setting takes around the same time (per party) as in the non-distributed setting. Indeed, this is what we expect because the computation that takes place is exactly the same as in the non-distributed setting, except that it happens to take place on shares rather than the actual values itself. Hence, according to these numbers, the cost of privacy is essentially that the computation is outsourced to three different workers, that each have to perform the same work as one worker in the non-private setting. Finally, as expected, verification time completely vanishes compared to computation time.

Our performance numbers should be interpreted as estimates. Our Pinocchio performance is around 8–9 times worse than in [27]; but on the other hand, we could not use their proprietary elliptic curve and pairing implementations; and we did not spend much time optimising performance. Note that, as expected, our Pinocchio and Trinocchio implementations have approximately the same running time. If Trinocchio would be based on Pinocchio's code base, we would expect the same. Moreover, apart from combining the proofs from different workers, the verification routines of Pinocchio and Trinocchio are exactly the same, so achieving faster verification than native computation as in [27] should be possible with Trinocchio as well. We also note that VIFF is not known for its speed, so replacing VIFF with a different multiparty computation framework should considerably speed up the computation of f.

6 Discussion and Conclusion

In this paper, we have presented Trinocchio, a system that adds privacy to the Pinocchio verifiable computation scheme essentially at the cost of replicating the Pinocchio proof production algorithm at three (or more) servers. Trinocchio has the same correctness and security guarantees as Pinocchio; distributing the computation between $2\theta + 1$ workers gives privacy if at most θ of them are corrupted. We have shown in a case study that the overhead is indeed small.

As far as we are aware, our work is the first to deliver efficient verifiable computation (i.e., with cryptographic guarantees of correctness and practical verification times independent of the computation size) with privacy guarantees. Although privacy is only guaranteed if not too many of the workers are corrupt, the use of verifiable computation ensures that the outcome of the protocol cannot be manipulated by the workers. This allows us to hedge against an adversary being more powerful than anticipated in a real world scenario.

As discussed, existing verifiable computation constructions in the single-worker setting [17, 20, 16] use very expensive cryptography, while multiple-worker efforts to provide privacy [2] do not guarantee correctness if all workers are corrupted. In contrast, existing works from the area of multiparty computation [3, 30, **?**] deliver privacy and correctness guarantees, but have much less efficient verification.

A major limitation of Pinocchio-based approaches is that they assume trusted set-up of the (function-dependent) evaluation and verification keys. In the single-client setting, the client could perform this set-up itself, but in the multiple-client setting, it is less clear who should do this. In particular, whoever has generated the evaluation and verification keys can use the values used during key generation as a trapdoor to generate proofs of false statements. Even though key generation can likely be distributed using the same techniques we use to distribute proof production, it remains the case that all generating parties together know this trapdoor. Unfortunately, this seems inherent to the Pinocchio approach.

Our work is a first step towards privacy-preserving verifiable computation, and we see many promising directions for future work. Recent work in verifiable computation has extended the Pinocchio approach by making it easier to specify computations [5], and by adding access control functionality [1]. In future work, it would be interesting to see how these kind of techniques can be used in the Trinocchio setting. Also, recent work has focused on applying verifiable computation on large amounts of data held by the server (and possibly signed by a third party) [10]; assessing the impact of distributing the computation (in particular when aggregating information from databases from several parties) in this scenario is also an important future direction. It would also be interesting to base Trinocchio on the (much faster) Pinocchio codebase [27] and more efficient multiparty computation implementations, and see what kind of performance improvements can be achieved. Another interesting direction is to investigate the possibility of practical universally composable [7, 8] distributed verifiable computation; or to use the universal composability framework to obtain a more generic framework for combining multiparty computation with verifiable computation (even with only standalone guarantees).

Acknowledgements This work was supported in part through the FP7 programme under grant 609611 (PRACTICE) and through the H2020 programme under grant 643964 (SUPERCLOUD).

References

1. J. Alderman, C. Janson, C. Cid, and J. Crampton. Access Control in Publicly Verifiable Outsourced Computation. In *Proc. ASIACCS*, 2015.
2. P. Ananth, N. Chandran, V. Goyal, B. Kanukurthi, and R. Ostrovsky. Achieving Privacy in Verifiable Computation with Multiple Servers - Without FHE and without Pre-processing. In *Proceedings of PKC*, 2014.
3. C. Baum, I. Damgård, and C. Orlandi. Publicly Auditable Secure Multi-Party Computation. In *Proceedings of SCN*, 2014.
4. M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *Proceedings of STOC*, 1988.
5. E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Proceedings of CRYPTO*. 2013.
6. R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, 13:2000, 1998.
7. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. *IACR Cryptology ePrint Archive*, 2000:67, 2000.
8. R. Canetti, A. Cohen, and Y. Lindell. A simpler variant of universally composable security for standard multiparty computation. In *Proceedings of CRYPTO 2015*, pages 3–22, 2015.
9. H. Carter, C. Lever, and P. Traynor. Whitewash: outsourcing garbled circuit generation for mobile devices. In *Proceedings of the 30th Annual Computer Security Applications Conference, ACSAC 2014, New Orleans, LA, USA, December 8-12, 2014*, pages 266–275, 2014.
10. A. Chiesa, E. Tromer, and M. Virza. Cluster Computing in Zero Knowledge. In *Proceedings of EUROCRYPT*, 2015.
11. S. G. Choi, J. Katz, R. Kumaresan, and C. Cid. Multi-client non-interactive verifiable computation. In *TCC*, pages 499–518, 2013.
12. C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 253–270. IEEE Computer Society, 2015.
13. I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Proc. 3rd Theory of Cryptography Conference (TCC 2006)*, volume 3876, pages 285–304, Berlin, 2006. Springer-Verlag.
14. I. Damgård and J. B. Nielsen. Perfect Hiding and Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor. In *Proceedings of CRYPTO*, 2002.

15. S. de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming.* PhD thesis, Eindhoven University of Technology, 2012.
16. D. Fiore, R. Gennaro, and V. Pastro. Efficiently Verifiable Computation on Encrypted Data. In *Proceedings of CCS*, 2014.
17. R. Gennaro, C. Gentry, and B. Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of CRYPTO*, 2010.
18. R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Proceedings of EUROCRYPT*. 2013.
19. R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In *Proceedings of PODC*, 1998.
20. S. Goldwasser, Y. T. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Proceedings of STOC*, 2013.
21. S. Gordon, J. Katz, F.-H. Liu, E. Shi, and H.-S. Zhou. Multi-client verifiable computation with stronger security guarantees. In Y. Dodis and J. Nielsen, editors, *Theory of Cryptography*, volume 9015 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin Heidelberg, 2015.
22. J. Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *Proceedings of ASIACRYPT*, 2010.
23. T. P. Jakobsen, J. B. Nielsen, and C. Orlandi. A framework for outsourcing of secure computation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security, CCSW '14, Scottsdale, Arizona, USA, November 7, 2014*, pages 81–92, 2014.
24. S. Kamara, P. Mohassel, and B. Riva. Salus: a system for server-aided secure function evaluation. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 797–808, 2012.
25. S. Mitsunari. A Fast Implementation of the Optimal Ate Pairing over BN curve on Intel Haswell Processor. Cryptology ePrint Archive, Report 2013/362, 2013. `http://eprint.iacr.org/`.
26. P. Mohassel and M. K. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Proceedings of PKC*, 2006.
27. B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of S&P*, 2013.
28. A. Peter, E. Tews, and S. Katzenbeisser. Efficiently outsourcing multiparty computation under multiple keys. *IEEE Transactions on Information Forensics and Security*, 8(12):2046–2058, 2013.
29. B. Schoenmakers and P. Tuyls. Efficient Binary Conversion for Paillier Encrypted Values. In *Proceedings of EUROCRYPT*, 2006.
30. B. Schoenmakers and M. Veeningen. Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems. In *Proceedings of ACNS*, 2015. `http://eprint.iacr.org/2015/058`.
31. B. Schoenmakers, M. Veeningen, and N. de Vreede. Trinocchio: Privacy-preserving outsourcing by distributed verifiable computation. In *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings*, pages 346–366, 2016.

A Verifiable Computation from QAPs

In this section, we discuss the protocol for verifiable computation based on quadratic arithmetic programs from [18, 27].

A.1 Modelling Computations as Quadratic Arithmetic Programs

A quadratic arithmetic program, or QAP, is a way of encoding arithmetic circuits, and some more general computations, over a field $\mathbb{F}$ of prime order q. It is given by a collection of polynomials over $\mathbb{F}$.

Definition 1 ([27]). *A* quadratic arithmetic program *Q over a field $\mathbb{F}$ is a tuple $Q = (\{v_i\}_{i=0}^k, \{w_i\}_{i=0}^k, \{y_i\}_{i=0}^k, t)$, with $v_i, w_i, y_i, t \in \mathbb{F}[x]$ polynomials of degree $\deg v_i, \deg w_i, \deg y_i < \deg t = d$. The polynomial t is called the* target polynomial. *The* size *of the QAP is k; the* degree *is the degree d of t.*

In the remainder, for ease of notation, we adopt the convention that $x_0 = 1$.

Definition 2. *Let $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ be a QAP. A tuple $(x_1, \ldots, x_k)$ is a* solution *of Q if t divides $(\sum_{i=0}^k x_i v_i) \cdot (\sum_{i=0}^k x_i w_i) - (\sum_{i=0}^k x_i y_i) \in \mathbb{F}[x]$.*

In case t splits, i.e., $t = (x-\alpha_1) \cdot \ldots \cdot (x-\alpha_n)$, a QAP can be seen as a collection of rank-1 quadratic equations for $(x_1, \ldots, x_k)$; that is, equations $v \cdot w - y$ with $v, w, y \in \mathbb{F}[x_1, \ldots, x_k]$ of degree at most one. Namely, $(x_1, \ldots, x_k)$ is a solution of Q if t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, which means exactly that, for every α_j, $(\sum_i x_i v_i(\alpha_j)) \cdot (\sum_i x_i w_i(\alpha_j)) - (\sum_i x_i y_i(\alpha_j)) = 0$: that is, each α_j gives a rank-1 quadratic equation in variables $(x_1, \ldots, x_k)$. Conversely, a collection of d such equations (recall $x_0 \equiv 1$)

$$(v_0^j \cdot x_0 + \ldots + v_k^j \cdot x_k) \cdot (w_0^j \cdot x_0 + \ldots + w_k^j \cdot x_k) - (y_0^j \cdot x_0 + \ldots + y_k^j \cdot x_k)$$

can be turned into a QAP by selecting d distinct elements $\alpha_1, \ldots, \alpha_d$ in $\mathbb{F}$, setting target polynomial $t = (x - \alpha_1) \cdot \ldots \cdot (x - \alpha_d)$, and defining v_0 to be the unique polynomial of degree smaller than d for which $v_0(\alpha_j) = v_0^j$, etcetera.

A QAP is said to compute a function $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ if the remaining x_i give a solution exactly if the function is correctly evaluated.

Definition 3 ([27]). *Let $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ be a QAP, and let $f : \mathbb{F}^l \to \mathbb{F}^m$ be a function. We say that Q* computes *f if $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ $\Leftrightarrow \exists\ (x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

For any function f given by an arithmetic circuit, we can easily construct a QAP that computes the function f. Indeed, we can describe an arithmetic circuit as a series of rank-1 quadratic equations by letting each multiplication gate become one equation. Apart from circuits containing just addition and multiplication gates, we can also express circuits with some other kinds of gates directly as QAPs. For instance, [27] defines a "split gate" that converts a number a into its k-bit decomposition $a_1, \ldots, a_k$ with equations $a = a_1 + 2 \cdot a_2 + \ldots + 2^{k-1} \cdot a_k$, $a_1 \cdot (1 - a_1) = 0$, …, $a_k \cdot (1 - a_k) = 0$.

A.2 Proving Correctness of Computations

If QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ computes a function f, then a prover can prove that $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ by proving knowledge of values $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q, i.e., t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$. [27] gives a construction of a proof system which does exactly this. The proof system assumes discrete logarithm groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ with a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_3$ for which the $(4d+4)$-PDH, d-PKE and $(8d+8)$-SDH assumptions [27] hold, with d the degree of the QAP. Moreover, the proof is in the common reference string (CRS) model: the CRS consists of an *evaluation key* used to produce the proof, and a *verification key* used to verify it. Both are public, i.e., provers can know the verification key and vice versa.

To prove that t divides $p = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, the prover computes quotient polynomial $h = p/t$ and basically provides evaluations "in the exponent" of h, $(\sum_i x_i v_i)$, $(\sum_i x_i w_i)$, and $(\sum_i x_i y_i)$ in an unknown point s that can be verified using the pairing. More precisely, given generators g_1 of $\mathbb{G}_1$ and g_2 of $\mathbb{G}_2$ (written additively) and polynomial $f \in \mathbb{F}[x]$, let us write $\langle f \rangle_1$ for $g_1 \cdot f(s)$ and $\langle f \rangle_2$ for $g_2 \cdot f(s)$. The evaluation key in the CRS, generated using random $s, \alpha_v, \alpha_w, \alpha_y, \beta, r_v, r_w, r_y = r_v \cdot r_w \in \mathbb{F}$, is:

$$\begin{gathered}\langle r_v v_i \rangle_1, \langle r_v \alpha_v v_i \rangle_1, \langle r_w w_i \rangle_2, \langle r_w \alpha_w w_i \rangle_1, \langle r_y y_i \rangle_1, \langle r_y \alpha_y y_i \rangle_1, \\ \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1, \langle s^j \rangle_1.\end{gathered}$$

where i ranges over $l+m+1, l+m+2, \ldots, k$ and j runs from 0 to the degree of t. The proof contains the following elements:

$$\begin{gathered}\langle V_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_v v_i \rangle_1 \cdot x_i, \quad \langle \alpha_v V_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot x_i, \\ \langle W_{\text{mid}} \rangle_2 = \textstyle\sum_i \langle r_w w_i \rangle_2 \cdot x_i, \quad \langle \alpha_w W_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot x_i, \\ \langle Y_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_y y_i \rangle_1 \cdot x_i, \quad \langle \alpha_y Y_{\text{mid}} \rangle_1 = \textstyle\sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot x_i, \\ \langle Z \rangle_1 = \textstyle\sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot x_i, \quad \langle H \rangle_1 = \textstyle\sum_j \langle s^j \rangle_1 \cdot h_j,\end{gathered} \tag{1}$$

where i ranges over $l+m+1, l+m+2, \ldots, k$, and h_j are the coefficients of polynomial $h = p/t$.

To verify that t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ and hence $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$, a verifier uses the following verification key from the CRS:

$$\langle \alpha_v \rangle_2, \langle \alpha_w \rangle_2, \langle \alpha_y \rangle_2, \langle \beta \rangle_1, \langle \beta \rangle_2, \langle r_v v_i \rangle_1, \langle r_w w_i \rangle_2, \langle r_y y_i \rangle_1, \langle r_y t \rangle_2,$$

where i ranges over $0, 1, 2, \ldots, l+m$[5]. Given the verification key, a proof, and values $x_1, \ldots, x_{l+m}$, the verifier proceeds as follows. First, it checks that

$$\begin{gathered}e(\langle V_{\text{mid}} \rangle_1, \langle \alpha_v \rangle_2) = e(\langle \alpha_v V_{\text{mid}} \rangle_1, \langle 1 \rangle_2); \\ e(\langle \alpha_w \rangle_1, \langle W_{\text{mid}} \rangle_2) = e(\langle \alpha_w W_{\text{mid}} \rangle_1, \langle 1 \rangle_2); \\ e(\langle Y_{\text{mid}} \rangle_1, \langle \alpha_y \rangle_2) = e(\langle \alpha_y Y_{\text{mid}} \rangle_1, \langle 1 \rangle_2) :\end{gathered} \tag{2}$$

[5] In [27], several terms of the verification key includes a value γ; however, a careful look at [27]'s proof reveals that γ is actually not needed. We remove it because it simplifies notation, especially for our multi-client protocols.

intuitively, under the d-PKE assumption, these checks guarantee that the prover must have constructed $\langle V_{\mathrm{mid}}\rangle_1$, $\langle W_{\mathrm{mid}}\rangle_2$, and $\langle Y_{\mathrm{mid}}\rangle_1$ using the elements from the evaluation key. It then checks that

$$e(\langle V_{\mathrm{mid}}\rangle_1 + \langle Y_{\mathrm{mid}}\rangle_1, \langle\beta\rangle_2) \cdot e(\langle\beta\rangle_1, \langle W_{\mathrm{mid}}\rangle_2) = e(\langle Z\rangle_1, \langle 1\rangle_2) : \tag{3}$$

under the PDH assumption, this guarantees that the same coefficients x_i were used in $\langle V_{\mathrm{mid}}\rangle_1$, $\langle W_{\mathrm{mid}}\rangle_2$, and $\langle Y_{\mathrm{mid}}\rangle_1$. Finally, the verifier computes evaluations $\langle V\rangle_1$ of $\sum_{i=0}^{k} x_i v_i$ as $\langle V_{\mathrm{mid}}\rangle_1 + \sum_{i=0}^{l+m} \langle r_v v_i\rangle_1 \cdot x_i$; $\langle W\rangle_2$ of $\sum_{i=0}^{k} x_i w_i$ as $\langle W_{\mathrm{mid}}\rangle_2 + \sum_{i=0}^{l+m} \langle r_w w_i\rangle_2 \cdot x_i$; and $\langle Y\rangle_1$ of $\sum_{i=0}^{k} x_i y_i$ as $\langle Y_{\mathrm{mid}}\rangle_1 + \sum_{i=0}^{l+m} \langle r_y y_i\rangle_1 \cdot x_i$, and verifies that

$$e(\langle V\rangle_1, \langle W\rangle_2) \cdot e(\langle Y\rangle_1, \langle 1\rangle_2)^{-1} = e(\langle H\rangle_1, \langle r_y t\rangle_2) : \tag{4}$$

under the $(8d+8)$-SDH assumption, this guarantees that, for the polynomial h encoded by $\langle H\rangle_1$, $t \cdot h = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ holds.[6]

Theorem 3 ([18], informal). *Given QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ and values $x_1, \ldots, x_{l+m}$, the above is a non-interactive argument of knowledge of $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

A.3 Making the Proof Zero-Knowledge

The above proof can be turned into a zero-knowledge proof, that reveals nothing about the values of $(x_{l+m+1}, \ldots, x_k)$ other than that t divides $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$ for some h, by performing randomisation. Namely, instead of proving that $t \cdot h = (\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$, we prove that $t \cdot \tilde{h} = (\sum_i x_i v_i + \delta_v \cdot t) \cdot (\sum_i x_i w_i + \delta_w \cdot t) - (\sum_i x_i y_i + \delta_y \cdot t)$ with $\delta_v, \delta_w, \delta_y$ random from $\mathbb{F}$. Precisely, the evaluation key needs to contain additional elements:

$$\langle r_v t\rangle_1, \langle r_v \alpha_v t\rangle_1, \langle r_w t\rangle_2, \langle r_w \alpha_w t\rangle_1, \langle r_y t\rangle_1, \langle r_y \alpha_y t\rangle_1, \langle r_v \beta t\rangle_1, \langle r_w \beta t\rangle_1, \langle r_y \beta t\rangle_1, \langle t\rangle_1.$$

Compared to the original proof, we let

$$\begin{aligned}
\langle V'_{\mathrm{mid}}\rangle_1 &= \langle V_{\mathrm{mid}}\rangle_1 + \langle r_v t\rangle_1 \cdot \delta_v, \quad \langle \alpha_v V'_{\mathrm{mid}}\rangle_1 = \langle \alpha_v V'_{\mathrm{mid}}\rangle_1 + \langle r_v \alpha_v t\rangle_1 \cdot \delta_v, \\
\langle W'_{\mathrm{mid}}\rangle_2 &= \langle W_{\mathrm{mid}}\rangle_2 + \langle r_w t\rangle_2 \cdot \delta_w, \quad \langle \alpha_w W'_{\mathrm{mid}}\rangle_1 = \langle \alpha_w W_{\mathrm{mid}}\rangle_1 + \langle r_w \alpha_w t\rangle_1 \cdot \delta_w, \\
\langle Y'_{\mathrm{mid}}\rangle_1 &= \langle Y_{\mathrm{mid}}\rangle_1 + \langle r_y t\rangle_1 \cdot \delta_y, \quad \langle \alpha_y Y'_{\mathrm{mid}}\rangle_1 = \langle \alpha_y Y_{\mathrm{mid}}\rangle_1 + \langle r_y \alpha_y t\rangle_1 \cdot \delta_y, \\
\langle Z'\rangle_1 &= \langle Z\rangle_1 + \langle r_v \beta t\rangle_1 \cdot \delta_v + \langle r_w \beta t\rangle_1 \cdot \delta_w + \langle r_y \beta t\rangle_1 \cdot \delta_y, \langle H'\rangle_1 = \textstyle\sum_j \langle s^j\rangle_1 \cdot \tilde{h}_j,
\end{aligned}$$

with $\tilde{h}_j$ the coefficients of $h + \delta_v w_0 + \sum_i \delta_v x_i \cdot w_i + \delta_w v_0 + \sum_i \delta_w x_i \cdot v_i + \delta_v \delta_w \cdot t - \delta_y$. Verification remains exactly the same.

Theorem 4 ([18], informal). *Given QAP $Q = (\{v_i\}, \{w_i\}, \{y_i\}, t)$ and values $x_1, \ldots, x_{l+m}$, the above is a non-interactive zero-knowledge argument of knowledge of $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q.*

[6] We remark that, as shown in [27], a verifier who has generated the evaluation and verification keys, can use the randomness from the generation process to save several of the above pairing checks. We do not consider this optimisation here.

A.4 From Arguments of Knowledge to Verifiable Computation

In [27], the above argument of knowledge is used to construct a *public verifiable computation scheme*. In such a scheme, a client outsources the computation of a function f to a worker, obtaining cryptographic guarantees that the result it gets from the worker is correct. It is defined as follows:

Definition 4 ([27]). *A* public verifiable computation scheme $\mathcal{VC}$ *consists of three polynomial-time algorithms* $(\mathsf{KeyGen}, \mathsf{Compute}, \mathsf{Verify})$*:*

- $(EK_f; VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda)$*: a probabilistic* key generation algorithm *that takes as argument a function* $f : \mathbb{F}^l \rightarrow \mathbb{F}^m$ *and a security parameter* λ*, outputting a public evaluation key* EK_f *and a public verification key* VK_f
- $(\boldsymbol{y}; \pi) \leftarrow \mathsf{Compute}(EK_f; \boldsymbol{x})$*: a probabilistic* worker algorithm *that takes input* $\boldsymbol{x} \in \mathbb{F}^l$ *and outputs* $\boldsymbol{y} = f(\boldsymbol{x}) \in \mathbb{F}^k$ *and a proof* π *of its correctness*
- $\{0, 1\} \leftarrow \mathsf{Verify}(VK_f; \boldsymbol{x}; \boldsymbol{y}; \pi)$*: a deterministic* verification algorithm *that outputs* 1 *if* $\boldsymbol{y} = f(\boldsymbol{x})$*, 0 otherwise.*

To outsource the computation of f, a client runs KeyGen and provides EK_f to the worker. When it needs $f(\boldsymbol{x})$, it provides $\boldsymbol{x}$ to the worker, who runs $\mathsf{Compute}$ and provides the result $\boldsymbol{y} = f(\boldsymbol{x})$ and proof π to the client. The client accepts $\boldsymbol{y}$ if Verify succeeds. We require that worker cannot provide incorrect proofs even if it knows VK_f, which makes this verifiable computation scheme "public". In fact, a trusted party could for once and for all perform KeyGen and publish $(\mathrm{EK}_f, \mathrm{VK}_f)$; any client who trusts this party can then use the published VK_f to verify computations. (Trusting this party is needed: the random values used in KeyGen are a trapdoor with which the generator of the keys can produce false proofs.) A public verifiable computation scheme should satisfy *correctness* and *security*. Correctness means that honest workers produce accepting proofs:

Definition 5 ([27]). *A public verifiable computation scheme* $\mathcal{VC}$ *is called* correct *if, for all* $f : \mathbb{F}^l \rightarrow \mathbb{F}^m$ *and* $\boldsymbol{x} \in \mathbb{F}$*:*

$$\begin{aligned}&\textit{if } (EK_f; VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\boldsymbol{y}; \pi) \leftarrow \mathsf{Compute}(EK_f; \boldsymbol{x}),\\ &\quad \textit{then } \mathsf{Verify}(VK_f; \boldsymbol{x}; \boldsymbol{y}; \pi) = 1.\end{aligned}$$

Security means that corrupt workers cannot convince clients of wrong results:

Definition 6 ([27]). *A public verifiable computation scheme* $\mathcal{VC}$ *is called* secure *if, for any* $f : \mathbb{F}^l \rightarrow \mathbb{F}^m$ *and probabilistic polynomial time adversary* $\mathcal{A}$*:*

$$\begin{aligned}&\mathsf{Pr}[\ (EK_f, VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\boldsymbol{x}; \boldsymbol{y}; \pi) \leftarrow \mathcal{A}(EK_f; VK_f) :\\ &\quad \boldsymbol{y} \neq f(\boldsymbol{x}) \wedge \mathsf{Verify}(VK_f; \boldsymbol{x}; \boldsymbol{y}; \pi) = 1\ ] = \mathsf{negl}(\lambda).\end{aligned}$$

Given a QAP Q that computes a function f, the argument of knowledge from Section A.2 directly gives a public verifiable computation scheme known as Pinocchio [27]: KeyGen is the computation of the evaluation and verification keys for Q; $\mathsf{Compute}$ computes $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$, $(x_{l+m+1}, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of Q, and proof (1); and Verify are the checks (2–4) for this proof.

Theorem 5 (Pinocchio [27], informal). *Let QAP Q be of degree d. Then the above construction is a secure and correct public verifiable computation scheme under the d-PKE, $(4d+4)$-PDH, and $(8d+8)$-SDH assumptions.*

B Multi-client Protocol and Security Proof

In this appendix we will give a more detailed description of the multi-client protocol of Section 4 and the security proof.

B.1 The Protocol

We now present our multi-client Trinocchio protocol in more detail. As before, we assume that each input party provides only a single input and each result party receives only a single output; that is, each block from Section 4.1 consists of only one wire. It should be clear from Section 4.1 how this can be generalised.

Communication Model and Notation We assume synchronous communication; pairwise secure channels between the input parties and workers; between the workers themselves; and between the workers and result parties. To ensure agreement between the parties about the inputs for the computation, we additionally assume a bulletin board. Through this bulletin board, parties can publish messages which can then be retrieved by any other party. Messages on the bulletin board are authenticated. In our protocol, we denote a party posting a message m as $\mathsf{Post}(m)$. For convenience, we don't explicitly denote a party retrieving information from the bulletin board; instead, we take $\mathsf{Post}(m)$ to mean that any party can now use the value for m.

Mixed Commitment Scheme We use a commitment scheme, which allows a party to commit to a certain value, without revealing that value to other parties, but, when at a later time this value is revealed, the other parties can be certain that the revealed value is equal to the original committed to value. Each party has its own public commitment key k and a commitment to a value v using randomness r is denoted $\mathsf{Commit}_k(v; r)$. Because, given explicit randomness, the commitment algorithm is deterministic, the commitment can be opened by simply revealing (v, r). Then any party can verify the commitment by simply recomputing it. To ensure input independence, the commitment scheme must be non-malleable. Each input party will produce one commitment, so each commitment key is used only once.

In particular, we use a so-called "mixed commitment scheme" [14]. In such a scheme, commitment keys can be generated in two ways. First, they can be generated such that the scheme is perfectly binding and computationally hiding, and a trapdoor exists with which the committed value can be extracted. Second, they can be generated such that the scheme is perfectly hiding and computationally binding, and a trapdoor exists with which commitments can be opened to any value. Moreover, the keys generated in the two ways should be computationally indistinguishable. In our protocol, commitment keys of the first, i.e., perfectly binding, kind are generated for all input parties by a trusted party (and the trapdoor thrown away), which we model by a hybrid call $k_1, \ldots, k_l \leftarrow \mathsf{ComGen}$. (In the simulator used for the security proof, commitment keys of the first kind are generated for corrupted input parties and commitment keys of the second kind are generated for honest input parties, with the trapdoors used when simulating the adversary.) Mixed commitments can be instantiated efficiently, e.g., using a cryptographic hash function in the random oracle model; or using Paillier encryption [14]: in this latter case, perfectly binding commitment keys are $k = (1+N)^r s^N \bmod N^2$, perfectly hiding commitment keys are $k = s^N \bmod N^2$, and commitments are $k^m u^N \bmod N^2$.

Overview of the Protocol Our protocol is shown as Algorithm 4. The protocol starts with hybrid calls to obtain the trusted commitment keys and Trinocchio evaluation and verification keys (lines 2–3). The remainder of the protocol consists of the *input phase* (lines 4–16), in which the input parties provide their inputs to the workers; the *computation phase*, in which the workers compute the function and Pinocchio proof (lines 17–31); and the *result phase*, in which the result parties obtain the output from the workers and verify its correctness (lines 32–41).

Input Phase In the input phase, each input party provides its input to the workers. Compared to the single-client case, in which the input party simply provided secret shares of its inputs, we need to take several additional steps. Namely, we need each input party to provide a block for its inputs that other parties can use to verify the proof; and we need to guarantee input independence, namely, that input parties cannot choose their inputs depending on those of others.

To achieve these goals, we proceed as follows. First, each input party computes a block for its input (line 5). Having each input party post its block on the bulletin board would break input independence (in effect, it binds the input parties who provide the blocks first). We circumvent this by letting each input party post a commitment to its block first (line 6). After all commitments have been posted, the input parties post the openings to the commitments, i.e., the blocks and commitment randomness (line 7). (This guarantees input independence because in the security proof, the inputs of the honest parties can still be changed after the corrupted parties provide their inputs.) After this, the validity of the commitments (line 9) and blocks (line 10) are checked; if any input party provided incorrect information, the computation is aborted. Note that ProofBlock used by the input parties could already be considered a commitment scheme [12], however, because of the way the CRS is constructed and used in the security proof for the protocol, we cannot make use of the trapdoor that would make it equivocable.

After the input blocks have been posted and checked, the inputs are provided to the workers in the form of $(2\theta, n)$ shares (line 11). The shared information is both input $[x_i]$ and block randomness $[\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}]$: the workers need this latter information to compute the proof's $\langle H \rangle_1$ element. Note that we use $(2\theta, n)$ shares: because $n = 2\theta + 1$, the shares of all workers recombine to a unique value and we do not need to worry about input parties handing out inconsistent shares. The workers check that the shares correspond to the broadcast block by computing additive shares of the block, posting them, and checking if their Shamir recombination (denoted by Combine) matches the value on the bulletin board (lines 13–15). Finally, the $(2\theta, n)$-shares are converted into (θ, n)-shares (each worker (θ, n)-shares its share and applies recombination a la [19]) used for the remainder of the computation (line 16).

Computation Phase In the computation phase, the workers compute function f, and produce a Pinocchio proof that this computation was performed correctly. The computation of f (line 17) and coefficients $\boldsymbol{H'}$ of the polynomial $h = (v \cdot w - y)/t$ (lines 18–21) are the same as in the single-client case. To generate the proof block for the internal wires, the workers first generate shared random values $[\![\delta_{v,\mathrm{mid}}]\!], [\![\delta_{w,\mathrm{mid}}]\!], [\![\delta_{y,\mathrm{mid}}]\!]$ (line 22): for instance, by letting each party share a random value or using pseudo-random secret sharing. They then call ProofBlock to produce the block using the shared wires and randomness (line 23). The blocks for the result parties are generated in the same way (lines 24–26). The coefficients of the randomised quotient polynomial $\boldsymbol{H}$ are computed from $\boldsymbol{H'}$ analogously to the zero-knowledge variant of Pinocchio (Appendix A.3); note that this requires computing overall randomness δ_v, δ_w, δ_y that is the sum of the randomness from all blocks in the proof. This gives $(2\theta, n)$ shares $[\langle H \rangle_1]$ of proof element $\langle H \rangle_1$ (line 30)

Having computed shares of all proof elements, the workers now post these shares on the bulletin board so that everybody can combine them to obtain the full proof. Note that the shares of individual workers might statistically depend on information that we do not want to reveal such as internal circuit wires. To avoid any problems because of this, the workers first re-randomise their proof elements by adding a new random sharing of zero; for instance, obtained by letting each worker share zero or using pseudo-random zero sharing (line 31).

Result Phase In the result phase, the result parties obtain their computation results, and verify them with respect to the information on the bulletin board. First, the result parties obtain secret shares of their output values, and the randomness used in their proof blocks (line 32). Then, they combine the values from the bulletin board into a full multi-client Pinocchio proof (lines 34–36), and verify this proof (lines 37–38). Finally, they recombine their output values (line 39), check if the secret shares of their output values correspond to the posted proof block (line 40), and output the computation result (line 41).

In this section we prove Theorem 2, i.e., we show that our multi-client Trinocchio protocol (Algorithm 4) correctly (always) and securely (if at most

Algorithm 4 Trinocchio: n-party verifiable computation

```
1: ▷ Input parties $\mathcal{I}$ have $x_i$, result parties $\mathcal{R}$ output $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$
2: parties $i \in \mathcal{I}$ do $(k_1, \ldots, k_n) \leftarrow \mathsf{ComGen}()$
3: parties $i \in \mathcal{I} \cup \mathcal{W} \cup \mathcal{R}$ do $(EK = (\{BK_i\}_i, \ldots), VK = (\{BV_i\}_i, \ldots)) \leftarrow \mathsf{MKeyGen}()$
4: parties $i \in \mathcal{I}$ do                                              ▷ input phase
5:     $(\delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \in_{\mathrm{R}} \mathbb{F}^3$; $\boldsymbol{Q}_i \leftarrow \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$
6:     sample commitment randomness $\rho_i$; $c_i \leftarrow \mathsf{Commit}_{k_i}(\boldsymbol{Q}_i; \rho_i)$; $\mathsf{Post}(c_i)$
7:     $\mathsf{Post}(\boldsymbol{Q}_i, \rho_i)$
8:     for all $j \in \mathcal{I} \setminus \{i\}$ do
9:         if $c_j \neq \mathsf{Commit}_{k_j}(\boldsymbol{Q}_j; \rho_j)$ then abort the protocol
10:        if $\mathsf{CheckBlock}(BV_j; \boldsymbol{Q}_j) = \perp$ then abort the protocol
11:    create $(2\theta, n)$-shares $([x_i], [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$ and distribute to the workers
12: parties $\mathcal{W}$ do
13:    for all $i \in \mathcal{I}$ do
14:        $[\boldsymbol{Q}_i] \leftarrow \mathsf{ProofBlock}(BK_i; [x_i]; [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$; $\mathsf{Post}([\boldsymbol{Q}_i])$
15:        if $\mathsf{Combine}([\boldsymbol{Q}_i]) \neq \boldsymbol{Q}_i$ then abort the protocol
16:        convert $(2\theta, n)$ shares $([x_i], [\delta_{v,i}], [\delta_{w,i}], [\delta_{y,i}])$ to $(\theta, n)$ shares $(\llbracket x_i \rrbracket, \ldots)$
17:    compute $(\llbracket x_{l+1} \rrbracket, \ldots, \llbracket x_k \rrbracket)$ using MPC                  ▷ computation phase
18:    $\llbracket \boldsymbol{v} \rrbracket \leftarrow \{(\sum_i v_i(\omega_j) \cdot \llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{V} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\llbracket \boldsymbol{v} \rrbracket)$; $\llbracket \boldsymbol{v}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\llbracket \boldsymbol{V} \rrbracket)$
19:    $\llbracket \boldsymbol{w} \rrbracket \leftarrow \{(\sum_i w_i(\omega_j) \cdot \llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{W} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\llbracket \boldsymbol{w} \rrbracket)$; $\llbracket \boldsymbol{w}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\llbracket \boldsymbol{W} \rrbracket)$
20:    $\llbracket \boldsymbol{y} \rrbracket \leftarrow \{(\sum_i y_i(\omega_j) \cdot \llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{Y} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\llbracket \boldsymbol{y} \rrbracket)$; $\llbracket \boldsymbol{y}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\llbracket \boldsymbol{Y} \rrbracket)$
21:    $[\boldsymbol{h}'] \leftarrow \{(\llbracket \boldsymbol{v}'_j \rrbracket \cdot \llbracket \boldsymbol{w}'_j \rrbracket - \llbracket \boldsymbol{y}'_j \rrbracket)/t(\Omega_j)\}_j$; $[\boldsymbol{H}'] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\boldsymbol{h}'])$
22:    $(\llbracket \delta_{v,\mathrm{mid}} \rrbracket, \llbracket \delta_{w,\mathrm{mid}} \rrbracket, \llbracket \delta_{y,\mathrm{mid}} \rrbracket) \in_{\mathrm{R}} \mathbb{F}^3$
23:    $\llbracket \boldsymbol{Q}_{\mathrm{mid}} \rrbracket \leftarrow \mathsf{ProofBlock}(BK_{\mathrm{mid}}; \llbracket x_{l+m+1} \rrbracket, \ldots, \llbracket x_k \rrbracket; \llbracket \delta_{v,\mathrm{mid}} \rrbracket, \llbracket \delta_{w,\mathrm{mid}} \rrbracket, \llbracket \delta_{y,\mathrm{mid}} \rrbracket)$
24:    for all $i \in \mathcal{R}$ do
25:        $(\llbracket \delta_{v,i} \rrbracket, \llbracket \delta_{w,i} \rrbracket, \llbracket \delta_{y,i} \rrbracket) \in_{\mathrm{R}} \mathbb{F}^3$
26:        $\llbracket \boldsymbol{Q}_i \rrbracket \leftarrow \mathsf{ProofBlock}(BK_i; \llbracket x_i \rrbracket; \llbracket \delta_{v,i} \rrbracket, \llbracket \delta_{w,i} \rrbracket, \llbracket \delta_{y,i} \rrbracket)$
27:    $[\delta_v] \leftarrow [\delta_{v,\mathrm{mid}}] + \sum_{i \in \mathcal{I} \cup \mathcal{R}} [\delta_{v,i}]$
28:    $[\delta_w] \leftarrow [\delta_{w,\mathrm{mid}}] + \sum_{i \in \mathcal{I} \cup \mathcal{R}} [\delta_{w,i}]$
29:    $[\delta_y] \leftarrow [\delta_{y,\mathrm{mid}}] + \sum_{i \in \mathcal{I} \cup \mathcal{R}} [\delta_{y,i}]$
30:    $[\boldsymbol{H}] \leftarrow [\boldsymbol{H}'] + \llbracket \delta_v \rrbracket \llbracket \boldsymbol{W} \rrbracket + \llbracket \delta_w \rrbracket \llbracket \boldsymbol{V} \rrbracket + \llbracket \delta_v \rrbracket \llbracket \delta_w \rrbracket \boldsymbol{T} - \llbracket \delta_y \rrbracket$; $[\langle H \rangle_1] \leftarrow \sum_{j=0}^{d} \langle s^j \rangle_1 [\boldsymbol{H}_j]$
31:    $\mathsf{Post}(\llbracket \boldsymbol{Q}_{\mathrm{mid}} \rrbracket + \llbracket 0 \rrbracket)$; $\mathsf{Post}([\langle H \rangle_1] + [0])$; for all $i \in \mathcal{R}$ do $\mathsf{Post}(\llbracket \boldsymbol{Q}_i \rrbracket + \llbracket 0 \rrbracket)$
32:    for all $i \in \mathcal{R}$ do send $(\llbracket x_i \rrbracket, \llbracket \delta_{v,i} \rrbracket, \llbracket \delta_{w,i} \rrbracket, \llbracket \delta_{y,i} \rrbracket)$ to res. party $i$   ▷ result phase
33: parties $i \in \mathcal{R}$ do
34:    for all $j \in \mathcal{R}$ do $\boldsymbol{Q}_j \leftarrow \mathsf{Combine}([\boldsymbol{Q}_j])$
35:    $\boldsymbol{Q} \leftarrow \mathsf{Combine}(\llbracket \boldsymbol{Q}_{\mathrm{mid}} \rrbracket) + \sum_{j \in \mathcal{I} \cup \mathcal{R}} \boldsymbol{Q}_j$
36:    $\langle H \rangle_1 \leftarrow \mathsf{Combine}([\langle H \rangle_1])$
37:    if $\mathsf{CheckBlock}(BV_{\mathrm{mid}}; \boldsymbol{Q}_{\mathrm{mid}}) = \perp \vee \exists j : \mathsf{CheckBlock}(BV_j; \boldsymbol{Q}_j) = \perp \vee$
38:      $\mathsf{CheckDiv}(VK; \boldsymbol{Q}; \langle H \rangle_1) = \perp$ then output $\perp$ and abort protocol
39:    $(x_i, \delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \leftarrow \mathsf{Combine}(\llbracket x_i \rrbracket, \llbracket \delta_{v,i} \rrbracket, \llbracket \delta_{w,i} \rrbracket, \llbracket \delta_{y,i} \rrbracket)$
40:    if $\boldsymbol{Q}_i \neq \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ then output $\perp$ and abort protocol
41:    output $x_i$
```

θ workers are passively corrupted) evaluates function f. Theorem 2 directly follows from Lemmas 1 and 2 below.

B.2 Trinocchio Correctly Evaluates f

To prove that Trinocchio correctly evaluates f, we construct a simulator that interacts with the trusted party for correct function evaluation shown in Figure 1. The simulator $\mathcal{S}_{\text{correct}}$ is given in Algorithm 5.

Lemma 1. *For every probabilistic polynomial-time adversary $\mathcal{A}$, $\mathcal{S}_{\text{correct}}$ is probabilistic polynomial time and the distribution ensembles* $\textsc{Exec}^{(\mathsf{ComGen},\mathsf{MKeyGen})}_{\mathsf{Trinocchio},\mathcal{A}}$ *and* $\textsc{CIdeal}_{f,\mathcal{S}_{correct}}$ *are computationally indistinguishable.*

Proof. To prove this lemma, we will start from the $\textsc{Exec}$ distribution ensemble and introduce increasingly modified distribution ensembles YAD_i, each indistinguishable from the next, to finally show that $\textsc{Exec}_{\mathsf{Trinocchio},\mathcal{A}}$ is computationally indistinguishable from $\textsc{Ideal}_{f,\mathcal{S}_{\text{correct}}}$. The simulator operates by simulating the protocol with respect to the given adversary $\mathcal{A}$, and finally returning whatever value the simulated adversary $\mathcal{A}$ returned. The lines in the simulator are labelled to explain which parts of the simulator mimic the real protocol, which are needed to interact with the ideal functionality, and which modifications are introduced and explained by the various YAD distributions.

The real protocol is aborted at several places if certain conditions are met. Note that this is always in response to checks on information on the bulletin board that anybody can perform, hence all protocol parties agree on whether the protocol is aborted. If the simulator follows the protocol and the protocol is aborted, the simulator sends $\perp$ to the ideal functionality on behalf of any corrupt input party whose input had not been sent yet, and proceeds to send $\emptyset$ as set of result parties to get the result, disregarding any messages it receives from the ideal functionality. It also completes the simulation of $\mathcal{A}$ to obtain its output. This ensures that the distribution $\textsc{Ideal}$ is well-defined for aborted protocols.

At various points, the simulator is instructed to terminate the simulation. This is not the same as aborting the simulated protocol. The simulation will be terminated whenever the simulator fails at some computation which is not part of the real protocol, but which is needed to achieve some security property, such as mimicking the real protocol. To terminate the simulation will mean that the output of the adversary in the ideal case will not be consistent with the output in the real case, i.e., it will signal an adversary that it is in fact operating in the ideal case. To show that the termination of the simulation does not enable the distinction between $\textsc{Exec}$ and $\textsc{Ideal}$, we will show below that each of the conditions which lead to termination of the simulated protocol can only occur with negligible probability.

We will now very briefly describe the purpose of each of the distributions YAD_i. Each consecutive pair of distributions is indistinguishable.

We now present the increasingly modified distributions YAD_i, every time showing indistinguishability between consecutive distributions.

Algorithm 5 Simulator $\mathcal{S}_{\text{correct}}(C, \{x_i\}_{i \in C}, z, \lambda)$ for correct function evaluation

```
1: ▷ $\mathcal{I}$: input parties, $\mathcal{W}$: workers, $\mathcal{R}$: result parties, $C$: corrupted parties
2: for all $i \in \mathcal{I}$ do
3:     if $i \in C$ then generate perfectly binding comm. key $k_i$, keep trapdoor ▷ $\mathsf{YAD}_1$
4:     else generate perfectly hiding commitment key $k_i$, keep trapdoor ▷ $\mathsf{YAD}_1$
5: generate modified $(EK = \{\{BK_i\}_i, \ldots\}, VK = \{\{BV_i\}_i, \ldots\})$ as in $\mathsf{YAD}_3$ ▷ $\mathsf{YAD}_3$
6: whenever the adversary queries $\mathsf{ComGen}$, return $(k_1, \ldots, k_l)$
7: whenever the adversary queries $\mathsf{MKeyGen}$, return $(EK, VK)$
8: on behalf of honest parties $i \in \mathcal{I}$ do
9:     sample commitment randomness $\rho'_i$ ▷ $\mathsf{YAD}_2$
10:    $c_i \leftarrow \mathsf{Commit}_{k_i}(\mathbf{0}; \rho'_i)$ ▷ $\mathsf{YAD}_2$
11:    $\mathsf{Post}(c_i)$ ▷ $\textsc{Exec}$
12: for all $i \in \mathcal{I} \cap C$ do
13:     extract $\hat{\boldsymbol{Q}}_i$ from $c_i$ using trapdoor ▷ $\mathsf{YAD}_2$
14:     if $\mathsf{CheckBlock}(BV_i; \hat{\boldsymbol{Q}}_i) = \perp$ then
15:         $x_i \leftarrow \perp$ ▷ $\mathsf{YAD}_2$
16:     else
17:         use the $d$-PKE extractor on $\hat{\boldsymbol{Q}}_i$ to obtain field elements $\delta_{v,i}$, $\delta_{w,i}$ and $\delta_{y,i}$
        and polynomials $V_i(x)$, $W_i(x)$ and $Y_i(x)$ of degree at most $d-1$; if the extractor
        fails, terminate the simulation ▷ $\mathsf{YAD}_4$
18:         set $x_i$ such that $V_i(x) = x_i v_i(x)$, $W_i(x) = x_i w_i(x)$ and $Y_i(x) = x_i y_i(x)$; if
        this is not possible, terminate the simulation ▷ $\mathsf{YAD}_5$
19:         send $x_i$ to the ideal functionality on behalf of corrupt input party $i$ ▷ $\textsc{Ideal}$
20: receive $\boldsymbol{x}$ from the ideal functionality ▷ $\textsc{Ideal}$
21: on behalf of honest parties $i \in \mathcal{I}$ do
22:     $(\delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \in_{\mathrm{R}} \mathbb{F}^3$ ▷ $\textsc{Exec}$
23:     $\boldsymbol{Q}_i \leftarrow \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ ▷ $\textsc{Exec}$
24:     create $\rho_i$ such that $c_i = \mathsf{Commit}_{k_i}(\boldsymbol{Q_i}; \rho_i)$ using trapdoor ▷ $\mathsf{YAD}_2$
25: simulate lines 7 through 39 of the real protocol on behalf of honest parties ▷ $\textsc{Exec}$
26: $F \leftarrow \emptyset$ ▷ $\textsc{Ideal}$
27: for all $i \in \mathcal{R} \setminus C$ do
28:     if $\boldsymbol{Q}_i \neq \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ then
29:         $F \leftarrow F \cup \{i\}$ ▷ $\textsc{Ideal}$
30: for all $\boldsymbol{Q}' \in \{\boldsymbol{Q}_{\text{mid}}\} \cup \{\boldsymbol{Q}_i\}_{i \in \mathcal{R} \cap C} \cup \{\boldsymbol{Q}_i\}_{i \in F}$ do
31:     use the $d$-PKE extractor on $\boldsymbol{Q}'$ to obtain field elements $\delta'_v$, $\delta'_w$ and $\delta'_y$ and
    polynomials $V'(x)$, $W'(x)$ and $Y'(x)$ of degree at most $d-1$; if the extractor fails,
    terminate the simulation ▷ $\mathsf{YAD}_4$
32:     set the corresponding entries in $\boldsymbol{x}$ such that $V'(x) = \sum_i x_i v_i(x)$, $W'(x) =$
    $\sum_i x_i w_i(x)$ and $Y'(x) = \sum_i x_i y_i(x)$, where $i$ ranges over the indices corresponding
    to the block $\boldsymbol{Q}'$ belongs to; if this is not possible, terminate simulation ▷ $\mathsf{YAD}_5$
33: if $t(x) \nmid (\sum_i x_i v_i(x))(\sum_i x_i w_i(x)) - \sum_i x_i y_i(x)$ then terminate sim. ▷ $\mathsf{YAD}_6$
34: Send $\mathcal{R} \setminus F$ to the ideal functionality ▷ $\textsc{Ideal}$
35: Return the output of the simulated adversary
```

YAD_1 The distribution ensemble YAD_1 is the $\textsc{Exec}$ distribution ensemble, where the set-up of the protocol is modified such that the commitment keys for the corrupt input parties are generated to be perfectly binding instead of perfectly hiding, and the simulator keeps the trapdoors. This distribution ensemble is computationally indistinguishable from $\textsc{Exec}_{\mathsf{Trinocchio},\mathcal{A}}$ based on the property of the mixed commitment scheme that the two kinds of commitment keys are indistinguishable.

YAD_2 For the distribution ensemble YAD_2, the protocol is further modified by producing commitments to 0 instead of the input proof blocks on behalf of the honest input parties. When the commitments are opened later in the protocol, the openings to correct proof blocks are created using the trapdoor information. Additionally, the proof blocks produced by corrupt input parties are extracted from their commitments, although the extracted blocks are not used any further at this stage.

Indistinguishability between YAD_2 and YAD_1 follows directly from the indistinguishability property of the commitment scheme. The commitment scheme also guarantees that commitments produced by the adversary can only be opened to the extracted proof block, i.e., that $\hat{\boldsymbol{Q}}_i = \boldsymbol{Q}_i$ for corrupt input parties i.

YAD_3 For distribution ensemble YAD_3, we will again modify the set-up of the protocol, but this time of the evaluation and verification keys. This happens analogously to [27]'s security proof. Instead of sampling s, α_v, α_w, α_y, r_v, r_w, β_{mid} and the β_i for $1 \leq i \leq l+m$ uniformly at random and generating the keys from these values, the set-up proceeds as follows.

For a given QAP of degree d, set $q \leftarrow 4d+4$, then sample $s \in_{\mathrm{R}} \mathbb{F}$. Next, set

$$\begin{aligned}\mathsf{chal} \leftarrow \{&\langle 1\rangle_1, \langle s\rangle_1, \langle s^2\rangle_1, \ldots, \langle s^q\rangle_1, \langle s^{q+2}\rangle_1, \ldots, \langle s^{2q}\rangle_1 \\ &\langle 1\rangle_2, \langle s\rangle_2, \langle s^2\rangle_2, \ldots, \langle s^q\rangle_2, \langle s^{q+2}\rangle_2, \ldots, \langle s^{2q}\rangle_2\}.\end{aligned}$$

From this point onwards, the value s will not be used directly to compute the keys. Instead, any key element derived from s will be generated from chal. This restriction will be necessary to complete the security proof later.

Randomly draw α_v, α_w, α_y, r'_v and r'_w. Also draw a random polynomial $\chi_{\text{mid}}(x)$ of degree at most $3d+3$ such that $\chi_{\text{mid}}(x)$ is of degree at most $3d+3$ and $\chi_{\text{mid}}(x) \cdot (r'_v v_i(x) + r'_w x^{d+1} w_i(x) + r'_v r'_w x^{2d+2} y_i(x))$ has a zero coefficient in front of x^{3d+3} for all internal wire indices i, and $\chi_{\text{mid}}(x)t(x)$, $\chi_{\text{mid}}(x)x^{d+1}t(x)$ and $\chi_{\text{mid}}(x)x^{2d+2}t(x)$ have a zero coefficient in front of x^{3d+3} as well. Such polynomials exist by Lemma 10 of [18]. Similarly, for each input and output wire $1 \leq i \leq l+m$, draw random polynomial $\chi_i(x)$ such that $\chi_i(x)$ is of degree at most $3d+3$ and $\chi_i(x) \cdot (r'_v v_k(x) + r'_w x^{d+1} w_k(x) + r'_v r'_w x^{2d+2} y_k(x))$, $\chi_i(x)t(x)$, $\chi_i(x)x^{d+1}t(x)$ and $\chi_i(x)x^{2d+2}t(x)$ have a zero coefficient in front of x^{3d+3}

Now, we will generate the evaluation and verification keys as if we had used the following

$$\begin{aligned} r_v &= r'_v s^{d+1} \\ r_w &= r'_w s^{2d+2} \\ r_y &= r'_y s^{3d+3} \\ \beta_{\text{mid}} &= s\chi_{\text{mid}}(s) \\ \beta_i &= s\chi_i(s), \end{aligned}$$

where i ranges from 1 to $l + m$. Because we are not allowed to inspect the value of s directly, we cannot compute these values explicitly. However, we can compute the evaluation and verification key elements from chal. Because r_v, r_w and various β's are still distributed uniformly, and $r_y = r_v \cdot r_w$ still holds, the distribution of the keys is statistically indistinguishable from keys generated using the real key generation algorithm.

YAD_4 Distribution ensemble YAD_4 is produced in the same manner as YAD_3, except that the d-PKE extractor is run on the adversarially generated proof blocks that satisfy the $\mathsf{CheckBlock}$ predicate. If the extractor fails then the simulation is terminated. Because the d-PKE assumption states that the probability of failure is negligible, YAD_4 will be statistically indistinguishable from YAD_3. Therefore an adversary cannot cause the simulation to fail with better than negligible probability in an attempt to distinguish EXEC from IDEAL and the use of the d-PKE extractor on lines 17 and 31 is justified.

YAD_5 In addition to extracting the contents of all proof blocks, to produce distribution ensemble YAD_5 we will also attempt to retrieve the $\boldsymbol{x}$ values that constitute the extracted $V(x)$, $W(x)$ and $Y(x)$ polynomials. If no $\boldsymbol{x}$ exists such that $V(x) = \sum_i x_i v_i(x)$, $W(x) = \sum_i x_i w_i(x)$ and $Y(x) = \sum_i x_i y_i(x)$, then the simulation is terminated. We will show that an adversary that successfully causes this failure, i.e., with higher than negligible probability, can break the q-PDH assumption, as in the security proof of [27].

Suppose an adversary manages to produce a proof block $\boldsymbol{Q}$, corresponding to block verification key BK for which $\mathsf{CheckBlock}(VK; \boldsymbol{Q})$ holds and $V(x)$, $W(x)$ and $Y(x)$, as well as δ_v, δ_w and δ_y are successfully extracted, but no $\boldsymbol{x}$ exists satisfying $V(x) = \sum_i x_i v_i(x)$, $W(x) = \sum_i x_i w_i(x)$ and $Y(x) = \sum_i x_i y_i(x)$. Let $\langle Z \rangle_1$ be the final element of $\boldsymbol{Q}$. Then we can write $\langle Z \rangle_1$ as a polynomial $\sum_i \xi_i x^i$

evaluated at s "in the exponent":

$$\begin{aligned}\langle Z\rangle_1 - &\langle r_v\beta t\rangle_1\delta_v + \langle r_w\beta t\rangle_1\delta_w + \langle r_y\beta t\rangle_1\delta_y\\ &= \sum_j \langle r_v\beta v_j + r_w\beta w_j + r_y\beta y_j\rangle_1 x_j\\ &= \langle s\chi(s)\cdot(r'_v s^{d+1}V(s) + r'_w s^{2d+2}W(s) + r'_v r'_w s^{3d+3}Y(s))\rangle_1\\ &= \langle \sum_i \xi_i x^i\rangle_1.\end{aligned}$$

By Lemma 10 of [18], the coefficient ξ_{q+1} for x^{q+1} is non-zero with high probability. We can then compute

$$\langle s^{q+1}\rangle_1 = \xi_{q+1}^{-1}\cdot(\langle Z\rangle_1 - \langle r_v\beta t\rangle_1\delta_v + \langle r_w\beta t\rangle_1\delta_w + \langle r_y\beta t\rangle_1\delta_y - \sum_i \xi_i\langle s^i\rangle_1)$$

using only information in the evaluation key.

Recall from YAD_3 that the very first step in generating this distribution ensemble is to create a q-PDH challenge for some secret value s and in the rest of the process any information derived from s is computed based on this challenge. If instead of generating the challenge ourselves, we consider it a given, then the algorithm for generating YAD_5 together with an adversary that successfully causes failure can as a whole be viewed as an algorithm that breaks the q-PDH assumption.

This justifies the extraction of all wire values from proof blocks on lines 18 and 32 of $\mathcal{S}_{\text{correct}}$.

YAD_6 Distribution ensemble YAD_6 is generated as YAD_5, except that if the divisibility check $\mathsf{CheckDiv}$ succeeds, we use the wire values obtained in the normal course of the protocol together with the wire values extracted in YAD_5 to test whether $t(x)$ truly divides $p(x) = (\sum_{i=0}^k x_i v_i(x))(\sum_{i=0}^k x_i w_i(x)) - \sum_{i=0}^k x_i y_i(x)$. If this is not the case then the simulation is terminated. We will show that the probability of an adversary forcing this failure is negligible, as an algorithm that successfully manages to cause such a failure can be used to break the $2q$-SDH assumption, closely following the security proof of [27].

Let $V(x) = \sum_{i=0}^k x_i v_i(x)$, $W(x) = \sum_{i=0}^k x_i w_i(x)$, and $Y(x) = \sum_{i=0}^k x_i y_i(x)$. Suppose that $t(x)$ does not divide $p(x) = V(x)W(x) - Y(x)$. Let r be a root of $t(x)$ but not of $p(x)$ and let $T(x) = t(x)/(x-r)$. Let $d(x) = \gcd(t(x), p(x))$ and $a(x)$ and $b(x)$ be polynomials of degree at most $2d-1$ and $d-1$ respectively such that $a(x)t(x) + b(x)p(x) = d(x)$. Set $A(x) = a(x)T(x)/d(x)$ and $B(x) = b(x)T(x)/d(x)$. These polynomials have no denominator since $d(x)$ divides $T(x)$. Then $A(x)t(x) + B(x)p(x) = T(x)$. Dividing by $t(x)$, we have $A(x) + B(x)p(x)/t(x) - 1/(x-r)$. Note that $\langle H\rangle_1 = \langle p/t\rangle_1$. We can now evaluate $\langle A\rangle_1$ and $\langle B\rangle_2$ using terms in the evaluation key. From these we can solve $e(\langle A\rangle_1, \langle 1\rangle_2)e(\langle H\rangle_1, \langle B\rangle_2) = e(\langle 1\rangle_1, \langle 1\rangle_2)^{1/(s-r)}$.

Note that the q-PDH challenge can be considered an incomplete $2q$-SDH challenge. If, as with YAD_5, we again do not generate the challenge ourselves, but consider it a given, the algorithm for generating YAD_6, along with an adversary that successfully causes failure can be viewed as an algorithm which break the $2q$-SDH assumption.

Ideal The distribution ensembles YAD_1 through YAD_6 are indistinguishable from each other and from $\textsc{Exec}$. Through the distribution ensembles YAD_1 to YAD_6, we have argued that the distribution of the adversary's interactions with real protocol parties are indistinguishable from its simulation by YAD_i. At the same time, the outputs of the honest result parties in each YAD_i are still according to the protocol. Comparing YAD_6 to $\textsc{Ideal}_{f,\mathcal{S}_{\text{correct}}}$, we see that the adversary's output is unchanged, but now honest result parties get the value computed by the trusted party instead of the value from the simulated protocol. However, note that if the simulation in YAD_6 is not terminated, then the vector $\boldsymbol{x}$ is in fact a solution to the QAP corresponding to inputs supplied to the trusted party. Hence, because the QAP computes f, the values from $\boldsymbol{x}$ that are output as computation results in YAD_6 are in fact the output of f on the inputs supplied to the trusted party. Therefore, the outputs of the honest result parties in YAD_6 and $\textsc{Ideal}$ are the same.

From Exec to Ideal Overall, the sequence of distribution ensembles shows that the real- and ideal-world executions of the protocol are computationally indistinguishable, hence the lemma follows. $\square$

B.3 Private Case

The simulator $\mathcal{S}_{\text{private}}$ for private function evaluation is given in Algorithm 6. We show that it works in situations when privacy is guaranteed:

Lemma 2. *For every probabilistic polynomial-time adversary $\mathcal{A}$ such that at most θ workers are passively corrupted, $\mathcal{S}_{\text{correct}}$ is probabilistic polynomial time and the distribution ensembles $\textsc{Exec}^{(\mathsf{ComGen},\mathsf{MKeyGen})}_{\mathsf{Trinocchio},\mathcal{A}}$ and $\textsc{Ideal}_{f,\mathcal{S}_{private}}$ are computationally indistinguishable.*

Proof. The simulator mostly runs the actual protocol, using zero inputs on behalf of honest parties. However, it needs to provide the inputs of the corrupted input parties to the trusted party, and make sure that corrupted result parties obtain the result from the trusted party. For the corrupted inputs, note that the simulator simulates at least $\theta + 1$ honest workers, hence it knows enough shares of the inputs of corrupted input parties to determine them and send them to the trusted party (lines 6–8). In order to manipulate the corrupted results, the simulator simulates normal Trinocchio key generation with respect to the adversary, but keeps trapdoor s (line 3). It can then use s to make sure that the proof block that was generated for the adversary during the protocol run indeed opens to the output value for the result party that the simulator gets from the trusted party (lines 11–18).

Algorithm 6 Simulator $\mathcal{S}_{\text{correct}}(C, \{x_i\}_{i \in C}, z, \lambda)$ for secure function evaluation

```
1: $\triangleright$ $\mathcal{I}$: input parties, $\mathcal{W}$: workers, $\mathcal{R}$: result parties, $C$: corrupted parties
2: Generate real commitment keys $k_1, \ldots, k_n$ as in the protocol; when $\mathcal{A}$ makes a
   hybrid call to $\mathsf{ComGen}$, return $k_1, \ldots, k_n$
3: Generate evaluation key $EK$ and verification key $VK$, keep trapdoor $s$; when $\mathcal{A}$
   makes a hybrid call to $\mathsf{MKeyGen}$, return $(EK, VK)$
4: for all $i \in \mathcal{I} \setminus C$ do $x_i \leftarrow 0$
5: Simulate lines 5 to 32 of the real protocol on behalf of honest input parties and
   workers. If the protocol aborts, send $\perp$ to the ideal functionality on behalf of corrupt
   input parties and abort the simulated protocol
6: for all $i \in \mathcal{I} \cap C$ do
7:     $x_i \leftarrow \mathsf{Combine}([\![x_i]\!])$
8:     Send $x_i$ to the ideal functionality on behalf of corrupt input party $i$
9: for all $i \in \mathcal{R} \cap C$ do
10:    Receive result $\hat{x}_i$ from the ideal functionality
11:    $\delta_{v,i} \leftarrow \mathsf{Combine}([\![\delta_{v,i}]\!])$
12:    $\delta_{w,i} \leftarrow \mathsf{Combine}([\![\delta_{w,i}]\!])$
13:    $\delta_{y,i} \leftarrow \mathsf{Combine}([\![\delta_{y,i}]\!])$
14:    $\hat{\delta}_{v,i} \leftarrow \delta_{v,i} + (x_i - \hat{x}_i)\frac{v_i(s)}{t(s)}$
15:    $\hat{\delta}_{w,i} \leftarrow \delta_{w,i} + (x_i - \hat{x}_i)\frac{w_i(s)}{t(s)}$
16:    $\hat{\delta}_{y,i} \leftarrow \delta_{y,i} + (x_i - \hat{x}_i)\frac{y_i(s)}{t(s)}$
17:    Create shares $([\![\hat{x}_i]\!], [\![\hat{\delta}_{v,i}]\!], [\![\hat{\delta}_{w,i}]\!], [\![\hat{\delta}_{y,i}]\!])$ such that they are consistent with the
   shares of $([\![x_i]\!], [\![\delta_{v,i}]\!], [\![\delta_{w,i}]\!], [\![\delta_{y,i}]\!])$ held by corrupt computation parties
18:    Send $([\![\hat{x}_i]\!], [\![\hat{\delta}_{v,i}]\!], [\![\hat{\delta}_{w,i}]\!], [\![\hat{\delta}_{y,i}]\!])$ to result party $i$
19: Return the output of the simulated adversary
```

To see that the Exec and Ideal distributions are the same, first note that because the workers are all semi-honest, the outputs of the result parties in Exec are always correct, and hence the same as in Ideal. Hence, we only have to worry about the observations made by the adversary.

Now, note that the simulator at no point uses, or even has access to, the honest input parties' private values. Since the simulator follows the real protocol specification up to line 32, the adversary cannot detect any deviations from the real protocol, other than might be caused by the fact that the input values for the honest parties do not match the distribution of real input values. However, the privacy properties of the underlying secure multiparty computation protocol imply that no data exchanged during the computation protocol reveals any information about the input or intermediate wire values. Moreover, the commitment scheme is used as in the protocol, so does not give the adversary chance of distinguishing the real and ideal world.

The only other information that the adversary learns is what is opened during the multiparty computation protocol, i.e., the shares of the proof blocks ($\boldsymbol{Q}$) and divisibility check term ($\langle H \rangle_1$). First, note that these shares reveal nothing more than the proof blocks and divisibility check term themselves, as these shares are freshly randomised using a zero sharing before they are revealed.

Now consider what the adversary learns from the proof blocks and divisibility check term. As observed in [18], the first, third and fifth elements of a proof block, $\langle V\rangle_1$, $\langle W\rangle_2$, and $\langle Y\rangle_1$, are uniformly distributed if the δ_v, δ_w and δ_y used to compute those are uniformly distributed as well. This holds regardless of which value $\boldsymbol{x}$ is used. Furthermore, once these three elements are known, the remaining four elements are fixed due to the verification relations. Because all of the proof blocks generated in the protocol are produce using randomly chosen values for δ_v, δ_w and δ_y, it holds that all proof blocks in the protocol are distributed uniformly randomly and do not reveal any information about the values they are composed from.

We conclude that the adversary sees no information that allows it to distinguish the real and ideal worlds, hence the lemma follows. $\square$