Trinocchio: Privacy-Friendly Outsourcing by Distributed Verifiable Computation

Berry Schoenmakers, Meilof Veeningen, and Niels de Vreede

Eindhoven University of Technology

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. Each worker essentially performs 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 two case studies.

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

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 [Gro10,GGPR13], Pinocchio [PHGR13] 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 [BSCG$^+$13], or adding access control [AJCC15].

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 on verifiable computation [GGP10], it has not been shown so far how it is efficiently achieved. Indeed, constructions in the single-worker setting rely on inefficient cryptographic primitives like fully homomorphic encryption [GGP10,FGP14] and functional encryption [GKP$^+$13]. 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.

However, by outsourcing a computation to multiple workers, it *is* possible to guarantee privacy (if not all workers are corrupted) and correctness. Indeed, [ACG$^+$14] presents a verifiable computation protocol combining privacy and correctness; but unfortunately, it guarantees neither privacy nor correctness if all workers are corrupted; and it places a much higher burden on the workers than, e.g., [PHGR13]. Alternatively, one can draw from work in multiparty computation: while this field usually considers the setting in which all parties engage in an interactive protocol, recent works [BDO14,SV15a,SV15b] have also considered outsourcing scenarios. In these works, correctness is always guaranteed; privacy only up to some threshold of number of corruptions. This is the best to be expected, because privacy *and* correctness if all workers are corrupted implies privacy and correctness in the single-worker scenario, which as noted is impractical. However, with these existing multiparty computation constructions, we 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 [PHGR13], but while also getting *privacy* against corrupted workers?

We answer this question in the affirmative by presenting Trinocchio. Trinocchio uses [PHGR13]-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. Moreover, the distribution of the proof is such that each individual worker performs essentially the same work as a normal Pinocchio prover in the non-distributed setting.

Trinocchio works not only in the single-client setting, but also in settings with multiple distrusting input and result parties. In the single-client scenario, for some computations we protect privacy even against (a limited number of) malicious workers; in the multi-client scenario, we only fully protect 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. Also, by offering privacy, Trinocchio might enable verifiable computation on sensitive information for which external processing is otherwise forbidden. We demonstrate the feasibility of our approach by implementing two case studies: we demonstrate Trinocchio's low overhead by repeating one of [PHGR13]'s case studies, and we use Trinocchio to improve both worker and client overhead in verifiable linear programming [SV15a].

Outline We first recap the Pinocchio protocol for verifiable computation based on quadratic arithmetic programs (Section 2). We then show how Trinocchio distributes the proof computation of Pinocchio in the single-client scenario, and define and prove security of the construction (Section 3). We generalise Trinocchio to the setting with multiple, mutually distrusting inputters and outputters (Section 4). Finally, we demonstrate the feasibility of Trinocchio by analysing its performance in two case studies: computing a multivariate polynomial evaluation and proving optimality of a linear programming solution (Section 5). We finish with a discussion and conclusions (Section 6).

2 Verifiable Computation from QAPs

In this section, we discuss the protocol for verifiable computation based on quadratic arithmetic programs from [GGPR13,PHGR13].

2.1 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 prime order field $\mathbb{F}$ of size q. It is given by a collection of polynomials over $\mathbb{F}$.

Definition 1 ([PHGR13]). *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 ([PHGR13]). *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, [PHGR13] defines a "split gate" that converts a number a into its k-bit decomposition $a_1, \ldots, a_k$ with equations $a = a_1 + 2 \cdot a_2 + \ldots + 2^{k-1} \cdot a_k$, $a_1 \cdot (1 - a_1) = 0$, ..., $a_k \cdot (1 - a_k) = 0$.

2.2 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 $(\sum_i x_i v_i) \cdot (\sum_i x_i w_i) - (\sum_i x_i y_i)$. [PHGR13] 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 [PHGR13] 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:

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

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

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

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

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

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

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

under the PDH assumption, this guarantees that the same coefficients x_i were used in $\langle V_{\text{mid}} \rangle_1$, $\langle W_{\text{mid}} \rangle_2$, and $\langle Y_{\text{mid}} \rangle_1$. Finally, the verifier computes evaluations $\langle V \rangle_1$ of $\sum_{i=0}^{k} x_i v_i$ as $\langle V_{\text{mid}} \rangle_1 + \sum_{i=1}^{l+m} \langle r_v v_i \rangle_1 \cdot x_i$; $\langle W \rangle_2$ of $\sum_{i=0}^{k} x_i w_i$ as $\langle W_{\text{mid}} \rangle_2 + \sum_{i=1}^{l+m} \langle r_w w_i \rangle_2 \cdot x_i$; and $\langle Y \rangle_1$ of $\sum_{i=0}^{k} x_i y_i$ as $\langle Y_{\text{mid}} \rangle_1 + \sum_{i=1}^{l+m} \langle r_y y_i \rangle_1 \cdot x_i$, 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.[1]

Theorem 1 ([GGPR13], 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.*

2.3 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:

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

[1] We remark that, as shown in [PHGR13], 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.

Compared to the original proof, we let

$$\begin{aligned}
\langle V'_{\text{mid}}\rangle_1 &= \langle V_{\text{mid}}\rangle_1 + \langle r_v t\rangle_1 \cdot \delta_v, \quad \langle \alpha_v V'_{\text{mid}}\rangle_1 = \langle \alpha_v V'_{\text{mid}}\rangle_1 + \langle r_v \alpha_v t\rangle_1 \cdot \delta_v, \\
\langle W'_{\text{mid}}\rangle_2 &= \langle W_{\text{mid}}\rangle_2 + \langle r_w t\rangle_2 \cdot \delta_w, \quad \langle \alpha_w W'_{\text{mid}}\rangle_1 = \langle \alpha_w W_{\text{mid}}\rangle_1 + \langle r_w \alpha_w t\rangle_1 \cdot \delta_w, \\
\langle Y'_{\text{mid}}\rangle_1 &= \langle Y_{\text{mid}}\rangle_1 + \langle r_y t\rangle_1 \cdot \delta_y, \quad \langle \alpha_y Y'_{\text{mid}}\rangle_1 = \langle \alpha_y Y_{\text{mid}}\rangle_1 + \langle r_y \alpha_y t\rangle_1 \cdot \delta_y,
\end{aligned}$$

$$\langle Z'\rangle_1 = \langle Z\rangle_1 + \langle r_v \beta t\rangle_1 \cdot \delta_v + \langle r_w \beta t\rangle_1 \cdot \delta_w + \langle r_y \beta t\rangle_1 \cdot \delta_y, \langle H'\rangle_1 = \textstyle\sum_j \langle s^j\rangle_1 \cdot \widetilde{h_j},$$

with $\widetilde{h_j}$ 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 2 ([GGPR13], 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.*

2.4 From Arguments of Knowledge to Verifiable Computation

In [PHGR13], 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 ([PHGR13]). *A* public verifiable computation scheme $\mathcal{VC}$ *consists of three polynomial-time algorithms* (KeyGen, Compute, Verify)*:*

- $(EK_f; VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda)$*: a probabilistic* key generation algorithm *that takes as argument a function $f : \mathbb{F}^l \to \mathbb{F}^m$ and a security parameter λ, outputting a public evaluation key EK_f and 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 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 ([PHGR13]). *A public verifiable computation scheme $\mathcal{VC}$ is called* correct *if, for all $f : \mathbb{F}^l \to \mathbb{F}^m$ 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 ([PHGR13]). *A public verifiable computation scheme $\mathcal{VC}$ is called* secure *if, for any $f : \mathbb{F}^l \to \mathbb{F}^m$ and probabilistic polynomial time adversary $\mathcal{A}$:*

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

Given a QAP Q that computes a function f, the argument of knowledge from Section 2.2 directly gives a public verifiable computation scheme known as Pinocchio [PHGR13]: 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 3 (Pinocchio [PHGR13], 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.*

3 Distributing the Prover Computation

Although the above public verifiable computation scheme guarantees that the client gets a correct result, it does not in any way protect the client's sensitive inputs against a curious worker. In this section, we remedy this by showing that the worker computation can be distributed between multiple workers with little overhead, in such a way that no single worker learns anything about the inputs or outputs to the computation.

3.1 Definitions

First, we adapt the definition of a public verifiable computation scheme to the distributed setting. In the original definition, after the one-time key setup KeyGen, a client would simply provide the function input to the worker, who would run $\mathsf{Compute}$, giving a result that was checked with Verify.

In the distributed setting, we replace the single worker by a set of n workers, who perform $\mathsf{Compute}$ by executing an interactive protocol. Because no single worker should see the function input, we cannot have the function input as argument for $\mathsf{Compute}$; instead, the client runs a $\mathsf{Distribute}$ algorithm that gives "representations" $\boldsymbol{i}$ that the client supplies to the respective workers, which they use as their input for $\mathsf{Compute}$. Similarly, the worker's output of $\mathsf{Compute}$ are "representations" $\boldsymbol{o}$ of the function output and proof. The client uses a $\mathsf{Combine}$ algorithm to determine the function output and proof from $\boldsymbol{o}$; and finally uses Verify to verify that the function output is correct:

Definition 7. *An n-*party public verifiable computation scheme *$\mathcal{VC} = (\mathsf{KeyGen}, \mathsf{Distribute}, \mathsf{Compute}, \mathsf{Combine}, \mathsf{Verify})$ consists of four polynomial-time algorithms KeyGen, $\mathsf{Distribute}$, $\mathsf{Combine}$, and Verify, and an n-party protocol $\mathsf{Compute}$:*

- $(EK_f; VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda)$*: a probabilistic* key generation algorithm *that takes as argument a function* $f : \mathbb{F}^l \to \mathbb{F}^m$ *and a security parameter* λ*, outputting a public evaluation key* EK_f *and verification key* VK_f
- $\boldsymbol{i} \leftarrow \mathsf{Distribute}(VK_f; \boldsymbol{x})$*: a probabilistic* input distribution algorithm *that takes input* $\boldsymbol{x}$ *and outputs a length-n list of representations* $\boldsymbol{i}$ *of the input for the respective workers*
- $\boldsymbol{o} \leftarrow \mathsf{Compute}(EK_f; \boldsymbol{i})$*: a probabilistic* worker protocol *between n workers such that each worker j has input* $EK_f, \boldsymbol{i}_j$ *and output* $\boldsymbol{o}_j$
- $(\boldsymbol{y}; \pi) \leftarrow \mathsf{Combine}(VK_f; \boldsymbol{o})$*: a deterministic* output combination algorithm *that takes the outputs* $\boldsymbol{o}$ *of the workers and combines them into function output* $\boldsymbol{y} \in \mathbb{F}^m$ *and proof* π
- $\{0, 1\} \leftarrow \mathsf{Verify}(VK_f; \boldsymbol{x}; \boldsymbol{y}; \pi)$*: the deterministic* verification algorithm *outputs* 1 *if* $\boldsymbol{y} = f(\boldsymbol{x})$*, 0 otherwise.*

For the purposes of this paper, the protocol $\boldsymbol{o} \leftarrow \mathsf{Compute}(\mathrm{EK}_f; \boldsymbol{i})$ is a collection of n polynomial time algorithms that communicate over secure, private channels; each algorithm has as input EK_f and $\boldsymbol{i}_j$ and produces output $\boldsymbol{o}_j$. We denote by $\mathrm{Exec}_{\mathsf{Compute},\mathcal{A}}(\mathrm{EK}_f; \boldsymbol{i})$ the result of executing this protocol with adversary $\mathcal{A}$ acting on behalf of a fixed set of corrupted parties. Depending on the adversary model, $\mathcal{A}$ may act semi-honestly, i.e., it sticks to the protocol, or maliciously, i.e., it can communicate whatever it wants. Apart from the corrupted workers' inputs, $\mathcal{A}$ also has access to VK_f. The output of $\mathrm{Exec}_{\mathsf{Compute},\mathcal{A}}$ consists of a vector $\boldsymbol{o}$ of outputs of the n workers ($\perp$ for corrupted workers); and a special value a output by the adversary. $\mathrm{Exec}_{\mathsf{Compute}}(\mathrm{EK}_f; \boldsymbol{i})$ denotes a protocol execution without adversary.

As in the non-distributed case, an n-party public verifiable computation scheme should satisfy correctness and security. Correctness states that if a computation is outsourced without any adversary, then it will succeed:

Definition 8. *An n-party public verifiable computation scheme* $\mathcal{VC}$ *is called* correct *if, for all* $f : \mathbb{F}^l \to \mathbb{F}^m$ *and* $\boldsymbol{x} \in \mathbb{F}^l$*:*

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

Security states that a verifying proof implies a correct function result. We require that it holds regardless of which workers the adversary corrupts:

Definition 9. *An n-party public verifiable computation scheme* $\mathcal{VC}$ *is called* secure *if, for all* $f : \mathbb{F}^l \to \mathbb{F}^m$ *and any probabilistic polynomial time adversary* $\mathcal{A}$ *controlling any number of workers:*

$$\begin{aligned} \mathsf{Pr}[\ &(EK_f, VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda);\\ &\boldsymbol{x} \leftarrow \mathcal{A}(EK_f, VK_f); \boldsymbol{i} \leftarrow \mathsf{Distribute}(VK_f; \boldsymbol{x});\\ &(\boldsymbol{o}; a) \leftarrow \mathit{Exec}_{\mathsf{Compute},\mathcal{A}}(EK_f; \boldsymbol{i}); (\boldsymbol{y}; \pi) \leftarrow \mathsf{Combine}(VK_f; \boldsymbol{o}) :\\ &\boldsymbol{y} \neq f(\boldsymbol{x}) \ \wedge \ \mathsf{Verify}(VK_f; \boldsymbol{x}; \boldsymbol{y}; \pi) = 1\ ] = \mathsf{negl}(\lambda). \end{aligned}$$

The new "privacy" property for n-party public verifiable computation means that an adversary controlling a certain number of workers learns nothing about the inputs of the client. We define this by means of an experiment, in which the adversary first chooses two sets of inputs $\boldsymbol{x}^0$, $\boldsymbol{x}^1$, and then based on his involvement in $\mathsf{Compute}$ has to decide which of them was used. Privacy means that the adversary in this game has only a negligible advantage. We say that a scheme is *θ-passively private* if privacy holds with respect to a semi-honest adversary $\mathcal{A}$ controlling up to θ workers; and *θ-actively private* if privacy even holds with respect to an active adversary that corrupts up to θ workers:

Definition 10. *An n-party public verifiable computation scheme $\mathcal{VC}$ is called θ-*passively private *(resp. θ-*actively private*) if, for all $f : \mathbb{F}^l \to \mathbb{F}^m$ and any probabilistic polynomial time adversary $\mathcal{A}$ semi-honestly (resp. actively) controlling at most θ parties:*

$$\begin{aligned} | \mathsf{Pr}[\ & (EK_f, VK_f) \leftarrow \mathsf{KeyGen}(f, 1^\lambda); (\boldsymbol{x}^0; \boldsymbol{x}^1) \leftarrow \mathcal{A}(EK_f; VK_f); \\ & b \in_R \{0,1\}; \boldsymbol{i} \leftarrow \mathsf{Distribute}(VK_f; \boldsymbol{x}^b); (\boldsymbol{o}; a) \leftarrow Exec_{\mathsf{Compute},\mathcal{A}}(EK_f; \boldsymbol{i}) : \\ & b = a\ ] - 1/2\ | = \mathsf{negl}(\lambda). \end{aligned}$$

3.2 Construction

We now present Trinocchio, an n-party public verifiable computation scheme that combines correctness, security (regardless of corruptions) and privacy against at most θ semi-honest workers, 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.

To distribute the Pinocchio computation, Trinocchio employs multiparty computation techniques based on Shamir secret sharing [BGW88]. 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

Algorithm 1 Trinocchio's Compute algorithm

```
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($\mathsf{EK}_f = \{\langle r_v v_i \rangle_1\}_i, \ldots, \{\langle s^j \rangle_1\}_j; \llbracket x_1 \rrbracket, \ldots, \llbracket x_l \rrbracket$)
4:     $(\llbracket x_{l+1} \rrbracket, \ldots, \llbracket x_k \rrbracket) \leftarrow f(\llbracket x_1 \rrbracket, \ldots, \llbracket x_l \rrbracket)$
5:     $\llbracket \boldsymbol{v} \rrbracket \leftarrow \{\sum_i v_i(\alpha_j) \cdot \llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{V} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{v})$; $\llbracket \boldsymbol{v}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\boldsymbol{V})$
6:     $\llbracket \boldsymbol{w} \rrbracket \leftarrow \{\sum_i w_i(\alpha_j) \cdot \llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{W} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{w})$; $\llbracket \boldsymbol{w}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\boldsymbol{W})$
7:     $\llbracket \boldsymbol{y} \rrbracket \leftarrow \{\sum_i y_i(\alpha_j) \cdot \llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{Y} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{y})$; $\llbracket \boldsymbol{y}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\boldsymbol{Y})$
8:     $[\boldsymbol{h}'] \leftarrow \{(\llbracket \boldsymbol{v}'_j \rrbracket \cdot \llbracket \boldsymbol{w}'_j \rrbracket - \llbracket \boldsymbol{y}'_j \rrbracket)/t(\beta_j)\}_j$; $[\boldsymbol{H}] \leftarrow \mathsf{FFT}_{\mathcal{T}}^{-1}([\boldsymbol{h}'])$
9:     $\llbracket \langle V_{\mathrm{mid}} \rangle_1 \rrbracket \leftarrow \sum_i \langle r_v v_i \rangle_1 \cdot \llbracket x_i \rrbracket$
10:    $\llbracket \langle \alpha_v V_{\mathrm{mid}} \rangle_1 \rrbracket \leftarrow \sum_i \langle r_v \alpha_v v_i \rangle_1 \cdot \llbracket x_i \rrbracket$
11:    $\llbracket \langle W_{\mathrm{mid}} \rangle_2 \rrbracket \leftarrow \sum_i \langle r_w w_i \rangle_2 \cdot \llbracket x_i \rrbracket$
12:    $\llbracket \langle \alpha_w W_{\mathrm{mid}} \rangle_1 \rrbracket \sum_i \langle r_w \alpha_w w_i \rangle_1 \cdot \llbracket x_i \rrbracket$
13:    $\llbracket \langle Y_{\mathrm{mid}} \rangle_1 \rrbracket \leftarrow \sum_i \langle r_y y_i \rangle_1 \cdot \llbracket x_i \rrbracket$
14:    $\llbracket \langle \alpha_y Y_{\mathrm{mid}} \rangle_1 \rrbracket \leftarrow \sum_i \langle r_y \alpha_y y_i \rangle_1 \cdot \llbracket x_i \rrbracket$
15:    $\llbracket \langle Z \rangle_1 \rrbracket \leftarrow \sum_i \langle r_v \beta v_i + r_w \beta w_i + r_y \beta y_i \rangle_1 \cdot \llbracket x_i \rrbracket$
16:    $[\langle H \rangle_1] = \sum_j \langle s^j \rangle_1 \cdot [\boldsymbol{H}_j]$
17:    return $(\llbracket x_{l+1} \rrbracket, \ldots, \llbracket x_{l+m} \rrbracket; \llbracket \langle V_{\mathrm{mid}} \rangle_1 \rrbracket, \llbracket \langle \alpha_v V_{\mathrm{mid}} \rangle_1 \rrbracket, \llbracket \langle W_{\mathrm{mid}} \rangle_2 \rrbracket, \llbracket \langle \alpha_w W_{\mathrm{mid}} \rangle_1 \rrbracket,$
18:                $\llbracket \langle Y_{\mathrm{mid}} \rangle_1 \rrbracket, \llbracket \langle \alpha_y Y_{\mathrm{mid}} \rangle_1 \rrbracket, \llbracket \langle Z \rangle_1 \rrbracket, [\langle H \rangle_1])$
```

due to [GRR98]. Interactive protocols for many other tasks such as comparing two shared value also exist (see, e.g., [dH12]).

In more detail, Trinocchio's KeyGen and Verify are the same as Pinocchio's. To perform Pinocchio's Compute in a distributed way, secret sharing is used. $(\boldsymbol{i}_1, \ldots, \boldsymbol{i}_n) \leftarrow \mathsf{Distribute}(\mathrm{VK}_f; x_1, \ldots, x_l)$ computes secret sharings $\llbracket x_1 \rrbracket, \ldots, \llbracket x_l \rrbracket$: $\boldsymbol{i}_j$ are the shares for the jth party. For Compute, each worker takes as input EK_f and its shares of $\llbracket x_1 \rrbracket, \ldots, \llbracket x_l \rrbracket$, and returns shares $(\llbracket x_{l+1} \rrbracket, \ldots, \llbracket x_{l+m} \rrbracket$ and $\llbracket \langle V_{\mathrm{mid}} \rangle_1 \rrbracket$, $\llbracket \langle \alpha_v V_{\mathrm{mid}} \rangle_1 \rrbracket$, $\llbracket \langle W_{\mathrm{mid}} \rangle_2 \rrbracket$, $\llbracket \langle \alpha_w W_{\mathrm{mid}} \rangle_1 \rrbracket$, $\llbracket \langle Y_{\mathrm{mid}} \rangle_1 \rrbracket$, $\llbracket \langle \alpha_y Y_{\mathrm{mid}} \rangle_1 \rrbracket$, $\llbracket \langle Z \rangle_1 \rrbracket$, $[\langle H \rangle_1])$ of the Pinocchio proof. Finally, Combine applies Lagrange interpolation to the shares $\llbracket x_{l+1} \rrbracket, \ldots, \llbracket x_{l+m} \rrbracket$ to obtain computation result $\boldsymbol{y} = (x_{l+1}, \ldots, x_{l+m})$ and on the shares $\llbracket \langle V_{\mathrm{mid}} \rangle_1 \rrbracket$, $\ldots$, $[\langle H \rangle_1])$ to obtain Pinocchio proof $\pi = (\langle V_{\mathrm{mid}} \rangle_1, \ldots, \langle H \rangle_1)$.

Trinocchio's Compute protocol is shown in more detail as Algorithm 1. The first step is to compute function output $(x_{l+1}, \ldots, x_{l+m}) = f(x_1, \ldots, x_l)$ and values $(x_{l+m+1}, \ldots, x_k)$ 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 [GRR98]. If f is represented by a circuit using more complicated gates, then specific protocols may be used: e.g., the split gate discussed in Section 2.1 can be evaluated using multiparty bit decomposition protocols [ST06]. 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 [BSCG+13], 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 $\mathcal{T} = \{\Omega_1, \ldots, \Omega_k\}$ (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, $\langle V_{\mathrm{mid}} \rangle_1, \ldots, \langle H \rangle_1$ 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 [PHGR13], this overhead is small. For each input and output, Verify includes three exponentiations, whereas Combine involves four additions and two multiplications; when using [PHGR13]'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 from Section 2.3 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 [BSCG+13]. There, it is noted that the operation $\boldsymbol{P} = \mathsf{FFT}_{\mathcal{S}}(\boldsymbol{p})$ and its inverse can be efficiently implemented if $\mathcal{S} = \{\omega, \omega^2, \ldots, \omega^d = 1\}$ 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, [BSCG+13] 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, [BSCG+13] remarks that for $\mathcal{T} = \{\eta\omega, \eta\omega^2, \ldots, \eta\omega^d = \eta\}$, 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

It is easy to see that correctness and security of Trinocchio both follow from the corresponding properties of Pinocchio; for details, see the appendix. Trinocchio's privacy depends on the the protocol being used to compute f in line 4 of Algorithm 1. A multiparty computation protocol is called *(statistically or computationally) θ-private* [Can98] if it correctly computes f and leaks no information whenever at most θ workers are passively corrupted. If such a protocol is used to compute f, then Trinocchio is θ-passively private.

Theorem 4. *Let $n = 2\theta + 1$, and suppose that a θ-private n-party protocol is used to compute function f in line 4 of Algorithm 1. Then Trinocchio is an n-party public verifiable computation scheme that is correct, secure, and θ-passively private assuming d-PKE, $(4d{+}4)$-PDH and $(8d{+}8)$-SDH with d the QAP degree.*

Moreover, if the protocol used to compute f also does not leak information in the event of an active attack, then Trinocchio satisfies privacy against active attackers. Namely, let us call a multiparty computation protocol *θ-actively private* if it does not leak any information to an active attacker controlling up to θ workers. For instance, the protocol from [GRR98] satisfies this notion as the attacker only learns θ many shares of any value.

Theorem 5. *Let $n = 2\theta + 1$, and suppose that a θ-actively private n-party protocol is used to compute f in line 4 of Algorithm 1. Then Trinocchio is an n-party public verifiable computation scheme that is correct, secure, and θ-actively private assuming d-PKE, $(4d{+}4)$-PDH and $(8d{+}8)$-SDH with d the QAP degree.*

The two theorems follow easily from the respective properties of the multiparty computation protocol used; see the appendix. Theorem 5 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, 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. [MF06]).

4 Handling Mutually Distrusting In- and Outputters

We now consider the scenario in which various parties wish to obtain the results of a computation applied to input held by other parties, who are willing to enable the computation, but not to divulge their private input values. 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 (Section 2.3), and we need to make sure that input parties choose their inputs independently from each other.

The parties obtaining the results are collectively referred to as the result parties $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$. Similarly, the parties providing their input to the computation are called input parties $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_l\}$. We implement this scenario by outsourcing the computation to workers $\mathcal{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_n\}$. Note that these sets of parties may be mutually disjoint, but are not required to be so.

4.1 Security Model

Because the multiple client scenario involves several different kinds of parties with respect to which we have to guarantee different security properties, security in this scenario is best defined using the ideal/real paradigm [Can98]. Indeed, this scenario can be seen as a generalisation of the usual scenario of secure multiparty computation, for which the ideal/real paradigm is the traditional technique used to define security.

The idea of the ideal/real paradigm is to define security by demanding that the outputs of the result parties and adversary in a protocol execution are distributed indistinguishable from those in an ideal world, in which the function is computed by an incorruptible third party. The information that this trusted party sends to the result parties and adversary determines under what conditions properties like correctness, privacy, and privacy hold. Note that in the multiple client scenario, an additionally relevant property is "input independence": i.e., we wish to ensure that input parties choose their inputs independently from others. (For instance, if $\mathcal{I}_2$ would be able to set its input to "$\mathcal{I}_1$'s input + 1" then this would not break privacy, but it is still undesirable.)

Exec denotes the probability distribution of real protocol executions. Let Π be a protocol between parties $\mathcal{I} \cup \mathcal{W} \cup \mathcal{R}$, i.e., a collection of polynomial time algorithms that communicate over secure, private channels. Let λ be a security parameter. Given $\boldsymbol{i} \in \mathbb{F}^l$, let $\mathrm{Exec}_{\Pi,\mathcal{A}}(\lambda; \boldsymbol{i})$ be the result of executing protocol Π with probabilistic polynomial-time adversary $\mathcal{A}$ adversary acting on behalf of a fixed set $C \subset \mathcal{I} \cup \mathcal{W} \cup \mathcal{R}$ of corrupted parties, of which $A \subset C$ are actively corrupted. Each honest input party $\mathcal{I}_j$ receives input $\boldsymbol{i}_j$, and the adversary receives the inputs of corrupted input parties; the honest parties and adversary then execute the protocol. $\mathrm{Exec}_{\Pi,\mathcal{A}}(\lambda; \boldsymbol{i})$ consists of a vector $\boldsymbol{o}$ of the outputs of the result parties ($\perp$ for corrupted result parties) and a special value a output by the adversary. To model access to trusted information, such as the evaluation and verification keys, we will allow parties to call certain trusted n-party functions $g_1, \ldots, g_j$ that are always evaluated correctly. An execution in this so-called $(g_1, \ldots, g_j)$-*hybrid model* [Can98,CDN01] is denoted $\mathrm{Exec}_{\Pi,\mathcal{A}}^{(g_1,\ldots,g_l)}(\lambda; \boldsymbol{i})$.

Similarly, Ideal denotes the probability distribution of ideal protocol executions. Let $\boldsymbol{i} \in \mathbb{F}^l$, let I be an algorithm giving the code of trusted party $\mathcal{T}$, and let $\mathcal{S}$ be a probabilistic polynomial-time adversary acting on behalf of a fixed set $C \subset \mathcal{I} \cup \mathcal{W} \cup \mathcal{R}$ of corrupted parties, of which $A \subset C$ are actively corrupted. Then $\mathrm{Ideal}_{I,\mathcal{S}}(\lambda; \boldsymbol{i})$ denotes the result of the following execution: honest input parties each send their input to $\mathcal{T}$; honest result parties receive their result from $\mathcal{T}$ and output this; the adversary $\mathcal{S}$ gets the input of corrupted input parties, can send arbitrary values on behalf of actively corrupted input parties, and receives the results of corrupted result parties. The trusted party $\mathcal{T}$ executes the code of I; hence, this code determines the security guarantees modelled by the ideal execution. The result of $\mathrm{Ideal}_{I,\mathcal{S}}(\lambda; \boldsymbol{i})$ is a vector $\boldsymbol{o}$ of the result parties' outputs ($\perp$ for corrupted result parties) and a special value a output by the adversary.

Our objective is to offer unconditional correctness of the computation results, but, as before, whether privacy can be guaranteed depends on which parties are corrupted and the level of corruption. In particular, as in the previous section, we will carry out the actual computation by means of a θ-private secure threshold n-party computation protocol. Such a protocol guarantees privacy as long as no worker is actively corrupt, and fewer than θ worker are passively corrupt. As a consequence, we will achieve privacy under the same conditions. Generalising Definition 10, we say that protocols in this scenario that provide privacy against θ passively secure workers are *θ-passively secure.* The corresponding ideal functionality $I_{\theta\text{-pvc}}$, which we explain below, is given in Algorithm 2.

Definition 11. *Protocol Π between parties $\mathcal{I} = \{\mathcal{I}_1, \ldots, \mathcal{I}_l\}$, $\mathcal{W} = \{\mathcal{W}_1, \ldots, \mathcal{W}_n\}$, and $\mathcal{R} = \{\mathcal{R}_1, \ldots, \mathcal{R}_m\}$ is called a θ-*passively secure n-party public verifiable computation protocol in the $(g_1, \ldots, g_j)$-hybrid model *if, for all probabilistic polynomial time adversaries $\mathcal{A}$ corrupting set $C \subset \mathcal{I} \cup \mathcal{W} \cup \mathcal{R}$ of parties and actively corrupting $A \subset C$, there exists a polynomial-time adversary $\mathcal{S}$ such that, for all $\boldsymbol{i} \in \mathbb{F}^l$:*

$$Exec_{\Pi,\mathcal{A}}^{(g_1,\ldots,g_j)}(\lambda; \boldsymbol{i}) \approx Ideal_{I_{\theta\text{-}pvc},\mathcal{S}}(\lambda; \boldsymbol{i}),$$

where $\approx$ denotes computational indistinguishability in security parameter λ.

If no trusted $g_1, \ldots, g_j$-calls are needed, we simply call the protocol a θ-passively secure n-party public verifiable computation protocol. θ-actively secure protocols, that also protect privacy against active attackers, are defined analogously.

We now discuss the ideal-world algorithm $I_{\theta\text{-pvc}}$ for θ-passively secure n-party public verifiable computation (Algorithm 2). The algorithm first retrieves the inputs from the respective parties (for corrupted input parties, the adversary controls what is sent). We guarantee privacy under some conditions: only if these conditions are not met, then the adversary receives the inputs (line 3–4). Furthermore, actively corrupt input parties are able to abort the protocol entirely, preventing all result parties from obtaining their outcome (represented in line 6), whereas actively corrupt workers are able to choose which result parties

Algorithm 2 $I_{\theta\text{-pvc}}$: θ-passively secure n-party public verifiable computation

```
1: function $I_{\theta\text{-PVC}}(A; C)$
2:     for each input party $i \in \mathcal{I}$, receive input $x_i$ from $i$
3:     if $\geq 1$ actively corrupt or $\geq \theta$ passively corrupt workers then
4:         send all inputs $\boldsymbol{x}$ to the adversary $\mathcal{S}$
5:     if any entry of $\boldsymbol{x}$ is equal to $\perp$ then
6:         set $\boldsymbol{r} \leftarrow \perp^{|\mathcal{R}|}$
7:     else
8:         compute $\boldsymbol{r} \leftarrow f(\boldsymbol{x})$
9:     if any worker is actively corrupt then
10:        receive a subset of result parties $F \subseteq \mathcal{R}$ from the adversary $\mathcal{S}$
11:        for each result party $i \in F$, set $r_i \leftarrow \perp$
12:    for each result party $i \in \mathcal{R}$, send its result $r_i$ to $i$
```

obtain their outcome and which do not (represented in step 11). However, if the protocol is not aborted, then the results are guaranteed to be correct (line 8), capturing the correctness and security properties of the protocol. Finally, note that the corrupt input parties have to provide their inputs before possibly seeing the honest parties' inputs (line 2); hence, independence of inputs is assured, even in the situation when privacy is not.

For ease of notation, we model that each input party provides a single input, and each result party receives a single output. Both our ideal functionality and our protocol are easily adapted to the more general case. Note that, to fully capture the notion of outsourced computation, we would also want to express that input and result parties only need to act at the beginning and the end of the protocol. However, because this is not a security property, it unfortunately cannot be captured by the ideal functionality.

4.2 Primitives

In addition to multiparty computation based on (θ, n) secret sharing as described in Section 3.2, our multi-client variant of Trinocchio uses several other primitives.

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 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.

In particular, we use a so-called "mixed commitment scheme" [DN02]. 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 $k_1, \ldots, k_l \leftarrow$ 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 Paillier encryption [DN02].

Bulletin Board To ensure agreement between the parties about the inputs for the computation, we use 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. We denote a party posting a message m as $\mathsf{Post}(m)$. (We consider the bulletin board part of the execution model, but it can also be seen as a hybrid call.) For convenience, we do not state when a party retrieves information from the bulletin board; instead, we assume that all parties have access at any time to all information that has been posted. We understand a bulletin board to be a somewhat more powerful primitive than a broadcast channel in that even after the completion of our protocol, any party can access the bulletin board and retrieve all information posted during the protocol.

4.3 Multi-Client Proofs and Keys

Our multi-client Trinocchio proofs are a generalisation of the zero-knowledge variant of Pinocchio from Section 2.3 with modified evaluation and verification keys. Recall that in Pinocchio, the proof terms $\langle V_{\mathrm{mid}} \rangle_1$, $\langle \alpha_v V_{\mathrm{mid}} \rangle_1$, $\langle W_{\mathrm{mid}} \rangle_2$, $\langle \alpha_w W_{\mathrm{mid}} \rangle_1$, $\langle Y_{\mathrm{mid}} \rangle_1$, $\langle \alpha_y Y_{\mathrm{mid}} \rangle_1$, 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 7 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 the $\mathsf{ProofBlock}$ procedure given in Algorithm 3. 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

Algorithm 3 The ProofBlock algorithm

```
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 $(\langle V\rangle_1, \langle V'\rangle_1, \langle W\rangle_2, \langle W'\rangle_1, \langle Y\rangle_1, \langle Y'\rangle_1, \langle Z\rangle_1)$
```

Algorithm 4 The CheckBlock algorithm

```
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 $\bot$
```

the wires. This is achieved by modifying Pinocchio's key generation procedure such that, instead of a sampling a single 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 the ProofBlock algorithm only performs linear operations on its $\boldsymbol{x}$, δ_v, δ_w and δ_y inputs. Therefore this procedure 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 output party, one block $\boldsymbol{Q}_{\mathrm{mid}} = (\langle V_{\mathrm{mid}}\rangle_1, \ldots, \langle Z_{\mathrm{mid}}\rangle_1)$ 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 the CheckBlock procedure defined in Algorithm 4. 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). In the protocol, the procedure that verifies this relation will be called CheckDiv. We denote the modified setup of the evaluation and verification keys by hybrid call KeyGen.

4.4 The Protocol

We now present our multi-client Trinocchio protocol. Like in the security model, we assume that each input party provides only a single input and each output party receives only a single output; that is, each block from Section 4.3 consists of only one wire. It should be clear from Section 4.3 how this can be generalised.

Our protocol is shown as Algorithm 5. 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 an *input phase* (lines 4–16), in which the input parties provide their inputs to the workers; a *computation phase*, in which the workers compute the function and Pinocchio proof (lines 17–31); and a *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 inputter provides its input to the workers. Compared to the single-client case, in which the inputter simply provided secret shares of its inputs, we need to take several additional steps. Namely, we need each inputter 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 inputters 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 inputters 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.

After the input blocks have been posted and checked, the inputs are provided to the workers in 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 inputters 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 $\mathsf{Combine}$) 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 [GRR98]) 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 Section 2.3; 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).

Output Phase In the output phase, the result parties obtain their computation results, and verify then 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).

Theorem 6. *The* Trinocchio *protocol is a θ-passively secure n-party public verifiable computation protocol, $n = 2\theta + 1$, in the* (ComGen, KeyGen)*-hybrid model assuming d-PKE, $(4d + 4)$-PDH and $(8d + 8)$-SDH with d the QAP degree.*

The proof of this Theorem (in the appendix) uses two simulators: one when privacy and correctness are guaranteed (i.e., with at most θ passively corrupted input parties), and another when only correctness is guaranteed. In the former case, we obtain privacy by simulating the multiparty computation of the proof with respect to the adversary without using honest inputs. In the latter case, we run the protocol together with the adversary: if this gives a fake Pinocchio proof, then one of the underlying problems can be broken.

5 Performance and Application to Certificate Validation

In this section, we show that our approach indeed adds privacy to verifiable computation with little overhead. We demonstrate this in two case studies. First, we take the "MultiVar Poly" application from [PHGR13], 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. Second, we show that, using Trinocchio, the performance of "verification by validation" due to [SV15a] can be considerably improved: in particular, we improve the client's performance by several orders of magnitude.

Algorithm 5 Trinocchio: n-party public verifiable computation

```
1: $\triangleright$ 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 $(EK = (\{BK_i\}_i,\ldots), VK = (\{BV_i\}_i,\ldots)) \leftarrow \mathsf{KeyGen}()$
4: parties $i \in \mathcal{I}$ do                                            $\triangleright$ 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:    $(\llbracket x_{l+1} \rrbracket,\ldots,\llbracket x_k \rrbracket) \leftarrow f(\llbracket x_1 \rrbracket,\ldots \llbracket x_l \rrbracket)$                 $\triangleright$ computation phase
18:    $\llbracket \boldsymbol{v} \rrbracket \leftarrow \{(\sum_i v_i(\omega_j)\cdot\llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{V} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{v})$; $\llbracket \boldsymbol{v}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\boldsymbol{V})$
19:    $\llbracket \boldsymbol{w} \rrbracket \leftarrow \{(\sum_i w_i(\omega_j)\cdot\llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{W} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{w})$; $\llbracket \boldsymbol{w}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\boldsymbol{W})$
20:    $\llbracket \boldsymbol{y} \rrbracket \leftarrow \{(\sum_i y_i(\omega_j)\cdot\llbracket x_i \rrbracket\}_j$; $\llbracket \boldsymbol{Y} \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{S}}^{-1}(\boldsymbol{y})$; $\llbracket \boldsymbol{y}' \rrbracket \leftarrow \mathsf{FFT}_{\mathcal{T}}(\boldsymbol{Y})$
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:    $\llbracket \boldsymbol{Q}_{\mathrm{mid}} \rrbracket \leftarrow \mathsf{ProofBlock}(BK_{\mathrm{mid}};\llbracket x_{l+1} \rrbracket,\ldots,\llbracket x_k \rrbracket;\llbracket \delta_{v,\mathrm{mid}} \rrbracket,\llbracket \delta_{w,\mathrm{mid}} \rrbracket,\llbracket \delta_{y,\mathrm{mid}} \rrbracket)$
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 $i$  $\triangleright$ output 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 $x_i$
```

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 4.1, this guarantees privacy against one passively corrupted worker, or, in the single-client case with a θ-actively private multiparty computation protocol, against one actively corrupted worker. 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[2] 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 [PHGR13], 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[3] of Trinocchio's Compute algorithm (Algorithm 1) that is split into two parts. The first part performs the evaluation of the function f (line 4), given as an arithmetic circuit, using the secret sharing implementation of VIFF[4]. (We use the arithmetic circuit produced by the Pinocchio compiler, hence f is exactly the same as in [PHGR13].) 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 [Mit13]'s implementation[5] of the discrete logarithm groups and pairings from [BSCG$^+$13].

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.

[2] Running Intel Xeon E5-2670 v2 Ivy Bridge with 4 GB SSD and 3.75 GiB RAM
[3] Implementation available at `<removed_for_blind_review>`
[4] In particular, the TUeVIFF variant (`http://www.win.tue.nl/~berry/TUeVIFF/`)
[5] See `https://github.com/herumi/ate-pairing`

	# 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)

Our performance numbers should be interpreted as estimates. Our Pinocchio performance is around 8–9 times worse than in [PHGR13]; 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 [PHGR13] 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.

5.2 Speeding Up Verification by Validation

In [SV15a], the idea is proposed to speed up verifiable outsourcing by exploiting the fact that, to see if a solution to a computation is correct, it is often not necessary to consider the whole circuit. Specifically, instead of proving that $\boldsymbol{y} = f(\boldsymbol{x})$, workers prove that $\phi(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y})$ holds for some predicate ϕ and "certificate" $\boldsymbol{a}$. [SV15a] proposes to use ElGamal encryptions and zero-knowledge proofs to prove $\phi(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{y})$. This gives feasible performance, although the overhead compared to just computing f is still quite large. We now show that using Trinocchio both reduces the worker effort and dwarfs the client effort.

Specifically, [SV15a] presents a case study in linear programming. Given a matrix $\boldsymbol{A} \in \mathbb{Z}^{m \times n}$ and vectors $\boldsymbol{b} \in \mathbb{Z}^m$, $\boldsymbol{c} \in \mathbb{Z}^n$, linear programming asks to find vector $\boldsymbol{x} \in \mathbb{Z}^n$ and quotient q such that $q > 0$; $\boldsymbol{x} \geq 0$; $\boldsymbol{A} \cdot \boldsymbol{x} \leq q \cdot \boldsymbol{b}$, and $(\boldsymbol{c} \cdot \boldsymbol{x})/q$ is minimal. (In multiparty computation, $\mathbb{Z}$ is embedded into a sufficiently large field $\mathbb{F}$.) To solve this problem, heavy iterative algorithms such as the simplex algorithm are needed; but given the so-called "dual solution" $\boldsymbol{p} \in \mathbb{Z}^m$ it is easy to verify that $\boldsymbol{x}$ is optimal by checking that $q > 0$; $\boldsymbol{p} \cdot \boldsymbol{b} = \boldsymbol{c} \cdot \boldsymbol{x}$; $\boldsymbol{A} \cdot \boldsymbol{x} \leq q \cdot \boldsymbol{b}$; $\boldsymbol{x} \geq 0$; $\boldsymbol{A} \cdot \boldsymbol{p} \leq q \cdot \boldsymbol{c}$; and $\boldsymbol{p} \leq 0$. This criterion can be formulated as a set of polynomial equations [SV15a], and, in fact, as a QAP, by formulating checks like $q > 0$ in terms of bit decompositions, e.g., $q - 1 = a_0 + a_1 \cdot 2 + a_2 \cdot 2^2 + \ldots$ and $a_0 \cdot (1 - a_0) = 1$, $a_1 \cdot (1 - a_1) = 0$, and so on.

We have adapted the simplex LP solver from [SV15a][6] to work over the field we need for our discrete logarithm and pairing groups; and then used Trinocchio's

[6] Taken from `http://meilof.home.fmf.nl/`

LP size	**blp**	**#it**	**Comp**	**Cert**	**Proof**	**Total**	**Ver**
5x5	31	4	89	4	8	102	0.07
20x20	40	9	289	23	52	364	0.07
48x70	34	25	1080	61	172	1314	0.07
48x70	65	48	2702	119	268	3090	0.07
103x150	61	62	5415	308	713	6436	0.07
288x202	93	176	48781	1257	2479	52516	0.06

Table 2. Performance of verifiable linear programming by validation with Trinocchio: bitlength of solution numbers, number of simplex iterations, time for computation, certificate computation, proof, and total; and verification time (all times in seconds)

Compute to produce the proof that the computed LP solution is optimal. Our performance numbers are shown in Table 2. As shown, producing the Pinocchio proof is only a small percentage of the total distributed computation, ranging from 5% to 13% of total computation time. (The percentage decreases with problem size. Asymptotically, the time needed to evaluate f is $O(m \cdot n \cdot (I + l))$, with $m \times n$ the dimensions of the LP, I the number of iterations and l the bitlength needed during the computation; proof time is $O(lmn \cdot \log lmn)$.) Verification is very fast, and in particular much faster than evaluating the simplex algorithm with VIFF's local execution mechanism (which takes 78s on the biggest LP).

Comparing our Trinocchio approach to the ElGamal-based proofs of [SV15a], our proofs are not only much faster to verify, but also faster to produce. For verification, [SV15a] report times that are two-thirds of proof time, where our verification time is almost constant (for our problem sizes, the dominant factor is the computation of the constantly many pairings) and also asymptotically much better, since it only depends linearly on the *sum* of the LP dimensions, and not at all on the bitlength. Concerning proof production, our measured times are comparable (even slightly better), but the circumstances are not. Namely, while [SV15a]'s experiments use machines comparable to ours, they measure local communication whereas we measure communication in the cloud. Since [SV15a]'s proof production requires significant communication, their running time in the cloud should be higher than reported in [SV15a]. (Although we could not verify this latter claim, we did find that running [SV15a]'s LP solver in the cloud is around four times slower than in [SV15a], which is probably for the same reason.) On the other hand, [SV15a]'s proof production has slightly better asymptotics: theirs has $O(lnm)$ running time compared to our $O(lnm \cdot \log lmn)$.

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 two case studies that the overhead is indeed small, and that applying Trinocchio leads to performance improvements for the “verifiability by certificate validation” paradigm of [SV15a].

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. As discussed, existing verifiable computation constructions in the single-worker setting [GGP10,GKP$^+$13,FGP14] use very expensive cryptography, while multiple-worker efforts to provide privacy [ACG$^+$14] do not guarantee correctness if all workers are corrupted. In contrast, existing works from the area of multiparty computation [BDO14,SV15b,SV15a] 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-friendly 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 [BSCG$^+$13], and by adding access control functionality [AJCC15]. 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) [CTV15]; 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. Finally, it would also be interesting to base Trinocchio on the (much faster) Pinocchio codebase [PHGR13] and more efficient multiparty computation implementations, and see what kind of performance improvements can be achieved.

References

[ACG$^+$14] 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.

[AJCC15] J. Alderman, C. Janson, C. Cid, and J. Crampton. Access Control in Publicly Verifiable Outsourced Computation. In *Proc. ASIACCS*, 2015.

[BDO14] C. Baum, I. Damgård, and C. Orlandi. Publicly Auditable Secure Multi-Party Computation. In *Proceedings of SCN*, 2014.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation. In *Proceedings of STOC*, 1988.

[BSCG+13] 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.

[Can98] R. Canetti. Security and Composition of Multi-party Cryptographic Protocols. *Journal of Cryptology*, 13:2000, 1998.

[CDN01] R. Cramer, I. Damgård, and J. Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. In *Proc. EUROCRYPT*. 2001.

[CTV15] A. Chiesa, E. Tromer, and M. Virza. Cluster Computing in Zero Knowledge. In *Proceedings of EUROCRYPT*, 2015.

[dH12] S. de Hoogh. *Design of large scale applications of secure multiparty computation: secure linear programming*. PhD thesis, Eindhoven University of Technology, 2012.

[DN02] I. Damgård and J. B. Nielsen. Perfect Hiding and Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor. In *Proceedings of CRYPTO*, 2002.

[FGP14] D. Fiore, R. Gennaro, and V. Pastro. Efficiently Verifiable Computation on Encrypted Data. In *Proceedings of CCS*, 2014.

[GGP10] R. Gennaro, C. Gentry, and B. Parno. Non-interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. In *Proceedings of CRYPTO*, 2010.

[GGPR13] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Proceedings of EUROCRYPT*. 2013.

[GKP+13] 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.

[Gro10] J. Groth. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *Proceedings of ASIACRYPT*, 2010.

[GRR98] 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.

[MF06] P. Mohassel and M. K. Franklin. Efficiency Tradeoffs for Malicious Two-Party Computation. In *Proceedings of PKC*, 2006.

[Mit13] 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/`.

[PHGR13] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *Proceedings of S&P*, 2013.

[ST06] B. Schoenmakers and P. Tuyls. Efficient Binary Conversion for Paillier Encrypted Values. In *Proceedings of EUROCRYPT*, 2006.

[SV15a] B. Schoenmakers and M. Veeningen. Guaranteeing Correctness in Privacy-Friendly Outsourcing by Certificate Validation. Cryptology ePrint Archive, Report 2015/339, 2015. `http://eprint.iacr.org/`.

[SV15b] B. Schoenmakers and M. Veeningen. Universally Verifiable Multiparty Computation from Threshold Homomorphic Cryptosystems. In *Proceedings of ACNS*, 2015. `http://eprint.iacr.org/2015/058`.

A Security in Single-Client Case

In this section, we discuss the correctness, security, and privacy properties of Trinocchio in the single-client case (Section 3).

For correctness of Trinocchio in the single-client case, note that together, the sequence of steps $\boldsymbol{i} \leftarrow \mathsf{Distribute}(\mathrm{VK}_f; \boldsymbol{x}); \boldsymbol{o} \leftarrow \mathrm{Exec}_{\mathsf{Compute}}(\mathrm{EK}_f; \boldsymbol{i}); (\boldsymbol{y}; \pi) \leftarrow \mathsf{Combine}(\mathrm{VK}_f; \boldsymbol{o})$ by construction gives the same result as the step $(\boldsymbol{y}; \pi) \leftarrow \mathsf{Compute}(\mathrm{EK}_f; \boldsymbol{x})$ from Pinocchio, hence correctness of Pinocchio directly implies correctness of Trinocchio. For security, suppose that Trinocchio's steps $\boldsymbol{x} \leftarrow \mathcal{A}(\mathrm{EK}_f, \mathrm{VK}_f)$; $\boldsymbol{i} \leftarrow \mathsf{Distribute}(\mathrm{VK}_f; \boldsymbol{x})$;$(\boldsymbol{o}; a) \leftarrow \mathrm{Exec}_{\mathsf{Compute},\mathcal{A}}(\mathrm{EK}_f; \boldsymbol{i})$; $(\boldsymbol{y}; \pi) \leftarrow \mathsf{Combine}(\mathrm{VK}_f; \boldsymbol{o})$ would lead to an incorrect proof being accepted. Note that none of these steps use any information that is not available to the Pinocchio attacker, hence performing the above steps gives an attacker breaking the security of Pinocchio.

Trinocchio's privacy depends on the the protocol being used to compute f in line 4 of Algorithm 1. A multiparty computation protocol is called *(statistically or computationally) θ-private* [Can98] if it correctly computes f and leaks no information whenever at most θ workers are passively corrupted. If such a protocol is used to compute f, then Trinocchio is θ-passively private. To see this, note that in the experiment of Definition 10, the attacker learns EK_f and VK_f (which do not depend on the bit b) and the information it gets from performing a multiparty computation of f: namely, the shares of the inputs as output by $\mathsf{Distribute}(\mathrm{VK}_f, \boldsymbol{x}^b)$ and its part of the protocol transcript of line 4 of Algorithm 1. However, if the protocol is θ-private, then this information from the $b = 0$ and $b = 1$ cases is (statistically or computationally) indistinguishable with respect to the $\mathcal{A}$, hence the attacker cannot distinguish $b = 0$ and $b = 1$ with non-negligible probability. Hence the Theorem from Section 3 follows:

Theorem 4. *Let $n = 2\theta + 1$, and suppose that a θ-private n-party protocol is used to compute function f in line 4 of Algorithm 1. Then Trinocchio is an n-party public verifiable computation scheme that is correct, secure, and θ-passively private assuming d-PKE, $(4d{+}4)$-PDH and $(8d{+}8)$-SDH with d the QAP degree.*

Moreover, if the protocol used to compute f also does not leak information in the event of an active attack, then Trinocchio satisfies privacy against active attackers. Namely, let us call a multiparty computation protocol *θ-actively private* if it does not leak any information to an active attacker controlling up to θ workers.[7] For instance, the protocol from [GRR98] satisfies this notion as the attacker only learns θ many shares of any value. Of course, the attacker might arbitrarily interfere with the computation, but as long as he does not learn any result of the computation, this does not give him any information. By the same reasoning as above, the other theorem from Section 3 follows:

[7] Note that this definition only makes sense if a single party both delivers the inputs and obtains the result. If a computation is passively secure but actively private, then actively corrupted workers can still arbitrarily manipulate the outcome of the computation. In particular, if they can do this in a controlled way, then the outcome of the computation is in effect the result of applying a different function on the inputs. A result party can learn this different output – but, if this party also delivers the input, then this is not a problem. The definition then means that the workers should learn no information about this different output.

Theorem 5. *Let $n = 2\theta + 1$, and suppose that a θ-actively private n-party protocol is used to compute f in line 4 of Algorithm 1. Then Trinocchio is an n-party public verifiable computation scheme that is correct, secure, and θ-actively private assuming d-PKE, $(4d+4)$-PDH and $(8d+8)$-SDH with d the QAP degree.*

B Security in Multi-Client Case

In this section we will prove Theorem 6, i.e., we show that our multi-client Trinocchio protocol (Algorithm 5) is secure according to the security definition given by our ideal functionality. To prove this theorem, we need to show that there exists an ideal-world simulator for every real-world adversary. Recall that our ideal functionality offers unconditional correctness of the protocol outcome; privacy, however, can only be guaranteed if no workers are actively corrupt and less than θ are passively corrupt. Because these two situations are very different, we prove the theorem by distinguishing these *private* and *correct* cases, and we will give separate simulators and prove their correctness for each case by two lemmas. The theorem directly follows from these two lemmas.

B.1 Correct

The simulator $\mathcal{S}_{\text{correct}}$ we use to prove our protocol secure when privacy is not guaranteed, i.e., more than the threshold workers are passively corrupt or any worker is actively corrupt, is given in Algorithm 6. We now prove that it works in this situation.

Lemma 1. *For all probabilistic polynomial time adversaries $\mathcal{A}$ corrupting any number of input and result parties, and actively corrupting at least one or passively corrupting over θ computation parties, and for all $\boldsymbol{x} \in \mathbb{F}^l$:*

$$Exec_{\mathsf{Trinocchio},\mathcal{A}}(\lambda; \boldsymbol{x}) \approx Ideal_{I_{\theta\text{-}pvc},\mathcal{S}_{correct}}(\lambda; \boldsymbol{x})$$

where $\approx$ denotes computational indistinguishability in security parameter λ.

To prove this lemma, we will start from the Exec distribution and introduce increasingly modified distributions YAD_i, each indistinguishable from the next, to finally show that $\text{Exec}_{\mathsf{Trinocchio},\mathcal{A}}(\lambda; \boldsymbol{x})$ is computationally indistinguishable from $\text{Ideal}_{I_{\theta\text{-pvc}},\mathcal{S}_{\text{correct}}}(\lambda; \boldsymbol{x})$. The simulator operates by simulating the protocol with respect to the given adversaery $\mathcal{A}$, 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 $F = \mathcal{R}$, disregarding any messages it receives from the ideal functionality. It also complete the simulation of $\mathcal{A}$ to obtain its output. This ensures that the distribution Ideal is well-defined in case the protocol is aborted.

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 Exec and 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 now present the increasingly modified distributions YAD_i, every time showing indistinguishability between consequitive distributions.

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

YAD_2 For the distribution 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 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 [PHGR13]'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.

Algorithm 6 The correct case simulator $\mathcal{S}_{\text{correct}}$

```
1: for all $i \in \mathcal{I}$ do
2:     if $i \in C$ then generate perfectly binding comm. key $k_i$, keep trapdoor ▷ $\mathsf{YAD}_1$
3:     else generate perfectly hiding commitment key $k_i$, keep trapdoor ▷ $\mathsf{YAD}_1$
4: generate modified $(EK = \{\{BK_i\}_i, \ldots\}, VK = \{\{BV_i\}_i, \ldots\})$ as in $\mathsf{YAD}_3$ ▷ $\mathsf{YAD}_3$
5: whenever the adversary queries $\mathsf{ComGen}$, return $(k_1, \ldots, k_l)$
6: whenever the adversary queries $\mathsf{KeyGen}$, return $(EK, VK)$
7: on behalf of honest parties $i \in \mathcal{I}$ do
8:     sample commitment randomness $\rho'_i$ ▷ $\mathsf{YAD}_2$
9:     $c_i \leftarrow \mathsf{Commit}_{k_i}(\mathbf{0}; \rho'_i)$ ▷ $\mathsf{YAD}_2$
10:    $\mathsf{Post}(c_i)$ ▷ Exec
11: for all $i \in \mathcal{I} \cap C$ do
12:     extract $\hat{\boldsymbol{Q}}_i$ from $c_i$ using trapdoor ▷ $\mathsf{YAD}_2$
13:     if $\mathsf{CheckBlock}(BV_i; \hat{\boldsymbol{Q}}_i) = \perp$ then
14:         $x_i \leftarrow \perp$ ▷ $\mathsf{YAD}_2$
15:     else
16:         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 ▷ $\mathsf{YAD}_4$
17:         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 ▷ $\mathsf{YAD}_5$
18:         send $x_i$ to the ideal functionality on behalf of corrupt input party $i$ ▷ Ideal
19: receive $\boldsymbol{x}$ from the ideal functionality ▷ Ideal
20: on behalf of honest parties $i \in \mathcal{I}$ do
21:     $(\delta_{v,i}, \delta_{w,i}, \delta_{y,i}) \in_{\mathrm{R}} \mathbb{F}^3$ ▷ Exec
22:     $\boldsymbol{Q}_i \leftarrow \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ ▷ Exec
23:     create $\rho_i$ such that $c_i = \mathsf{Commit}_{k_i}(\boldsymbol{Q_i}; \rho_i)$ using trapdoor ▷ $\mathsf{YAD}_2$
24: simulate lines 7 through 39 of the real protocol on behalf of honest parties ▷ Exec
25: $F \leftarrow \emptyset$ ▷ Ideal
26: for all $i \in \mathcal{R} \setminus C$ do
27:     if $\boldsymbol{Q}_i \neq \mathsf{ProofBlock}(BK_i; x_i; \delta_{v,i}, \delta_{w,i}, \delta_{y,i})$ then
28:         $F \leftarrow F \cup \{i\}$ ▷ Ideal
29: 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
30:     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 ▷ $\mathsf{YAD}_4$
31:     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 ▷ $\mathsf{YAD}_5$
32: 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. ▷ $\mathsf{YAD}_6$
33: Send $F$ to the ideal functionality ▷ Ideal
34: Return the output of the simulated aversary
```

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 $\chi_{\mathrm{mid}}(x)$ of degree at most $3d + 3$ such that $\chi_{\mathrm{mid}}(x)$ is of degree at most $3d + 3$ and $\chi_{\mathrm{mid}}(x) \cdot (r'_v v_i(x) + r'_w x^{d+1} w_i(x) + r'_v r'_w x^{2d+2} y_i(x))$ has a zero coefficient in front of x^{3d+3} for all internal wire indices i, and $\chi_{\mathrm{mid}}(x)t(x)$, $\chi_{\mathrm{mid}}(x)x^{d+1}t(x)$ and $\chi_{\mathrm{mid}}(x)x^{2d+2}t(x)$ have a zero coefficient in front of x^{3d+3} as well. Such polynomials exist by Lemma 10 of [GGPR13]. 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

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

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 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 16 and 30 is justified.

YAD_5 In addition to extracting the contents of all proof blocks, to produce distribution 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 [PHGR13].

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 [GGPR13], 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 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 procedure 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 17 and 31 of $\mathcal{S}_{\text{correct}}$.

YAD_6 Distribution 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 [PHGR13].

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 Through the distributions 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 $\mathrm{Ideal}_{I_{\theta\text{-pvc}}, \mathcal{S}_{\mathrm{correct}}}(\lambda; \boldsymbol{x})$, 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 result in YAD_6 are in fact the output of f on the inputs supplied to the trusted party. Hence also the outputs of the honest result parties in YAD_6 and Ideal are the same.

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

B.2 Private

The simulator $\mathcal{S}_{\mathrm{private}}$ for the private case is given in Algorithm 7. We show that it works in situations when privacy is guaranteed:

Lemma 2. *For all probabilistic polynomial time adversaries $\mathcal{A}$ corrupting any number of input and result parties, and passively corrupting at most θ computation parties, and for all $\boldsymbol{x} \in \mathbb{F}^l$:*

$$Exec_{\mathsf{Trinocchio},\mathcal{A}}(\lambda; \boldsymbol{x}) \approx Ideal_{I_{\theta\text{-}pvc}, \mathcal{S}_{private}}(\lambda; \boldsymbol{x})$$

where $\approx$ denotes computational indistinguishability in security parameter λ.

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

Algorithm 7 The private case simulator $\mathcal{S}_{\text{private}}$

```
1: Generate real commitment keys $k_1, \ldots, k_n$ as in the protocol; when $\mathcal{A}$ makes a hybrid call to $\mathsf{ComGen}$, return $k_1, \ldots, k_n$
2: Generate evaluation key $EK$ and verification key $VK$, keep trapdoor $s$; when $\mathcal{A}$ makes a hybrid call to $\mathsf{KeyGen}$, return $(EK, VK)$
3: for all $i \in \mathcal{I} \setminus C$ do $x_i \leftarrow 0$
4: 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
5: for all $i \in \mathcal{I} \cap C$ do
6:     $x_i \leftarrow \mathsf{Combine}([\![x_i]\!])$
7:     Send $x_i$ to the ideal functionality on behalf of corrupt input party $i$
8: for all $i \in \mathcal{R} \cap C$ do
9:     Receive result $\hat{x}_i$ from the ideal functionality
10:    $\delta_{v,i} \leftarrow \mathsf{Combine}([\![\delta_{v,i}]\!])$
11:    $\delta_{w,i} \leftarrow \mathsf{Combine}([\![\delta_{w,i}]\!])$
12:    $\delta_{y,i} \leftarrow \mathsf{Combine}([\![\delta_{y,i}]\!])$
13:    $\hat{\delta}_{v,i} \leftarrow \delta_{v,i} + (x_i - \hat{x}_i)\frac{v_i(s)}{t(s)}$
14:    $\hat{\delta}_{w,i} \leftarrow \delta_{w,i} + (x_i - \hat{x}_i)\frac{w_i(s)}{t(s)}$
15:    $\hat{\delta}_{y,i} \leftarrow \delta_{y,i} + (x_i - \hat{x}_i)\frac{y_i(s)}{t(s)}$
16:    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
17:    Send $([\![\hat{x}_i]\!], [\![\hat{\delta}_{v,i}]\!], [\![\hat{\delta}_{w,i}]\!], [\![\hat{\delta}_{y,i}]\!])$ to result party $i$
18: Return the output of the simulated aversary
```

obtain the result from the trusted party. For the corrupted inputs, note that the simulator controls at least $\theta+1$ computation parties, hence it knows enough shares of the inputs of corrupted input parties to determine them and send them to the trusted party (lines 5–7). In order to manipulate the corrupted results, the simulator simulates normal Trinocchio key generation with respect to the adversary, but keeps trapdoor s (line 2). 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 10–17).

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, are the information that is opened in the multipary 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 [GGPR13], 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.