Trinocchio: Privacy-Preserving Outsourcing by Distributed Verifiable Computation

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

[1] Department of Mathematics and Computer Science, TU Eindhoven, Eindhoven, The Netherlands
[2] Philips Research, Eindhoven, The Netherlands
meilof.veeningen@philips.com

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 highly efficient Pinocchio system was introduced as a major leap towards practical verifiable computation. Unfortunately, Pinocchio and other efficient verifiable computation systems require the client to disclose the inputs to the worker, which is undesirable for sensitive inputs. To solve this problem, we propose Trinocchio: a system that distributes Pinocchio to three (or more) workers, that each individually do not learn which inputs they are computing on. We fully exploit the almost linear structure of Pinochhio proofs, letting each worker essentially perform the work for a single Pinocchio proof; verification by the client remains the same. Moreover, we extend Trinocchio to enable joint computation with multiple mutually distrusting inputters and outputters and still very fast verification. We show the feasibility of our approach by analysing the performance of an implementation in a case study.

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.

M. Manulis et al. (Eds.): ACNS 2016, LNCS 9696, pp. 346–366, 2016.
DOI: 10.1007/978-3-319-39555-5_19

While privacy was already defined in the first paper to formalize verifiable computation [GGP10], it has not been shown so far how it is efficiently achieved, with existing constructions relying on efficient cryptographic primitives. By outsourcing a computation to multiple workers, it *is* possible to guarantee privacy (if not all workers are corrupted) and correctness, but existing constructions from the multiparty literature lose the most appealing feature of verifiable computation: namely, that computations can be verified very quickly, even in time independent from the computation size. This leads to the central question of this paper: can we perform verifiable computation with the *correctness* and *performance* guarantees of [PHGR13], but while also getting *privacy* against corrupted workers?

1.1 Our Contributions

In this paper, we introduce Trinocchio to show that indeed, it is possible to outsource a computation in a privacy-preserving way to multiple workers, while retaining the fast verification offered by verifiable computation. Trinocchio uses state-of-the-art [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. The critical observation is that the almost linear structure of Pinocchio proofs (supporting verification based on bilinear maps) allows us to distribute the computation of Pinocchio proofs such that individual workers perform essentially the same work as a normal Pinocchio prover in the non-distributed setting. Specifically, our contributions are:

- We show how to distribute the production of Pinocchio proofs in a privacy-preserving way to multiple workers, thereby achieving privacy-preserving verifiable computation in the setting with one client.
- We extend our system to settings with multiple distrusting input and result parties.
- We provide a precise security model capturing the security guarantees of our protocols: privacy, correctness, but also input independence.
- We demonstrate the practical feasibility of our approach by implementing a case study: we demonstrate Trinocchio's low overhead by repeating the multivariate polynomial evaluation case study of [PHGR13]'s.

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

1.2 Related Work

Privacy-preserving outsourcing to single workers has been considered in the literature, but constructions in this setting rely on inefficient cryptographic primitives like fully homomorphic encryption [GGP10, CKKC13, FGP14], functional encryption [GKP+13], and multi-input attribute-based encryption [GKL+15]. (This is not surprising: indeed, even without guaranteeing correctness, letting a single worker perform a computation on inputs it does not know would intuitively seem to require some form of fully homomorphic encryption.) Some of these works also consider a multi-client setting [CKKC13, GKL+15].

A large body of works considers multiparty computation for privacy-preserving outsourcing (see, e.g., [KMR12, PTK13, CLT14, JNO14]). These works do not consider verifiability and achieve correctness at best in the case that *all-but-one* workers are corrupt (due to inherent limitations of the underlying protocols). We stress that this is rather unsatisfactory for the outsourcing scenario, where one naturally wishes to cover the case that *all* workers are corrupt—dispensing of the need to trust any particular worker.

Concerning outsourcing to multiple workers, [ACG+14] presents a verifiable computation protocol combining privacy and correctness; but unfortunately, they guarantee neither privacy nor correctness if all workers are corrupted and may collude; and it places a much higher burden on the workers than, e.g., [PHGR13]. Alternatively, recent works [BDO14, dHSV16, SV15], like us, guarantee correctness independent of worker corruption, but privacy only under some conditions. Our work offers a substantial performance improvement over these works by fully exploiting a set-up that needs to be trusted both for guaranteeing privacy and for guaranteeing correctness.

We should mention that the notion of verifiability exists in various forms and the field has a richer background than presented here, however, we focus entirely on the notion of verifiable computation first formalized by [GGP10], because it is tailored to the outsourcing scenario.

1.3 Outline

We first briefly define the security model for privacy-preserving outsourced computation in Sect. 3. In Sect. 4, we show how Trinocchio distributes the proof computation of Pinocchio in the single-client scenario, and prove security of the construction. We generalise Trinocchio to the setting with multiple, mutually distrusting inputters and outputters in Sect. 5. Finally, we demonstrate the feasibility of Trinocchio in Sect. 6 by analysing its performance in a case study, computing a multivariate polynomial evaluation. We finish with a discussion and conclusions in Sect. 7.

For convenience, we also provide a brief overview of the Pinocchio protocol [PHGR13] for verifiable computation based on quadratic arithmetic programs in Sect. 2.

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 field $\mathbb{F}$ of prime order 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:

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

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

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

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

[1] In [PHGR13], several terms of the verification key includes a value γ; however, a careful look at [PHGR13]'s proof reveals that γ is actually not needed. We remove it because it simplifies notation, especially for our multi-client protocols.

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

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.[2]

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:

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

Compared to the original proof, we let

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

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.

[2] 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.

Secure function evaluation:

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

Correct function evaluation:

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

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

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*.*

3 Security Model for Privacy-Preserving Outsourcing

In this section, we define security for privacy-preserving outsourcing. Because we have interactive protocols between multiple parties (as opposed to a cryptographic scheme, like verifiable computation above), we define security using the ideal/real-paradigm [Can00a]. In our setting, the parties are several *result parties* that wish to obtain the result of a computation on inputs held by several *input parties*, who are willing to enable the computation, but not to divulge their private input values to anybody else. Therefore, they outsource the computation to several *workers*. (Input and result parties may overlap.) The simplest case is the “single-client scenario” in which one party is the single input/result party.

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

To define security, we will re-use the existing definition framework for secure function evaluation [Can00a]. These definitions not specific to the outsourcing setting; but the outsourcing setting will become apparent when we claim that a protocol, e.g., implements secure function evaluation *if at most X workers are corrupted.* Secure function evaluation is the problem to evaluate $(y_1, \ldots, y_m) = f(x_1, \ldots, x_m)$ with m parties such that the ith party inputs x_i and obtains y_i, and no party learns anything else. (In outsourcing, result parties have non-empty output, input parties have non-empty inputs, and workers have empty in- and outputs.) A protocol π *securely evaluates function f* if the outputs of the parties and adversary $\mathcal{A}$ in a real-world execution of the protocol can be emulated by the outputs of the parties and an adversary $\mathcal{S}_{\mathcal{A}}$ in an idealised execution, where f is computed by a trusted party that acts as shown in Fig. 1. Security is guaranteed because the trusted party correctly computes the function. Privacy is guaranteed because the adversary in the idealised execution does not learn anything it should not. Secure evaluation also implies *input independence*, meaning that an input party cannot let its input depend on that of another, e.g., by copying the input of another party; this is guaranteed because the adversary needs to provide the inputs of corrupted parties without seeing the honest inputs. Typically, protocols achieve secure function evaluation for a given, restricted class of adversaries, e.g., adversaries that are passive and only corrupt a certain number of workers. Protocols can require set-up assumptions; these are captured by giving protocol participants access to a set of functions $g_1, \ldots, g_k$ that are always evaluated correctly. In this case, we say that the protocol securely evaluates the function *in the $(g_1, \ldots, g_k)$-hybrid model.* For details, see [Can00a].

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

4 Distributing the Prover Computation

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

4.1 Multiparty Computation Using Shamir Secret Sharing

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

4.2 The Trinocchio Protocol

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

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

Algorithm 1 shows in detail how the secret-shares of the function output and Pinocchio proof are computed. The first step is to compute function output $x_2 = f(x_1)$ and values $(x_3, \ldots, x_k)$ such that $(x_1, \ldots, x_k)$ is a solution of the QAP (line 4). This is done using normal multiparty computation protocols based on secret sharing. If function f is represented by an arithmetic circuit, then it is evaluated

Algorithm 1. Trinocchio's Compute protocol

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

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 Sect. 2.1 can be evaluated using multiparty bit decomposition protocols [DFK+06,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, $\langle V_{\text{mid}} \rangle_1, \ldots, \langle H \rangle_1$ 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 (Sect. 2.3) and the techniques from Sect. 5. 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 $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_3$. 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.

4.3 Security of Trinocchio

Theorem 3. *Let f be a function. Let $n = 2\theta + 1$ be the number of workers used. Let d be the degree of the QAP computing f used in the Trinocchio protocol. Assuming the d-PKE, $(4d+4)$-PDH, and $(8d+8)$-SDH assumptions:*

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

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

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

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

Privacy Against Active Attacks. We remark that actually, Trinocchio in some cases provides privacy against corrupted workers as well. Namely, suppose that the protocol used to compute f does not leak any information to corrupted workers in the event of an active attack (even though in this case it may not guarantee correctness). For instance, this is the case for the protocol from [GRR98]: the attacker can manipulate the shares that it sends, which makes the computation

Algorithm 2. ProofBlock

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

return incorrect results; but since the attacker always learns only θ many shares of any value, it does not learn any information. Because the attacker learns no additional information from producing the Pinocchio proof, the overall protocol still leaks no information to the adversary. (And security of Pinocchio ensures the client notices the attacker's manipulation.)

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

5 Handling Mutually Distrusting In- and Outputters

We now consider the scenario where there are multiple (possibly overlapping) input and result parties. There are some significant changes between this scenario and the single-client scenario. In particular, we need to extend Pinocchio to allow verification not based on the actual input/output values (indeed, no party sees all of them) but on some kind of representation that does not reveal them. Moreover, we need to use the zero-knowledge variant of Pinocchio (Sect. 2.3), and we need to make sure that input parties choose their inputs independently from each other.

5.1 Multi-client Proofs and Keys

Our multi-client Trinocchio proofs are a generalisation of the zero-knowledge variant of Pinocchio (Sect. 2.3) with modified evaluation and verification keys. Recall that in Pinocchio, the proof terms $\langle V_{\text{mid}}\rangle_1$, $\langle \alpha_v V_{\text{mid}}\rangle_1$, $\langle W_{\text{mid}}\rangle_2$, $\langle \alpha_w W_{\text{mid}}\rangle_1$, $\langle Y_{\text{mid}}\rangle_1$, $\langle \alpha_y Y_{\text{mid}}\rangle_1$, and $\langle Z\rangle_1$ encode circuit values $x_{l+m+1}, \ldots, x_k$; in the zero-knowledge variant, these terms are randomised so that they do not reveal any information about $x_{l+m+1}, \ldots, x_k$. In the multi-client case, additionally, the inputs of all input parties and the outputs of all result parties need to be encoded such that no other party learns any information about them. Therefore, we extend the proof with *blocks* of the above seven terms for each input and result party, which are constructed in the same way as the seven proof terms above. Although some result parties could share a block of output values, for simplicity we assign each result party its own block in the protocol.

Algorithm 3. CheckBlock

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

To produce a block containing values $\boldsymbol{x}$, a party first samples three random field values δ_v, δ_w, and δ_y and then executes ProofBlock, cf. Algorithm 2. The BK argument to this algorithm is the *block key*; the subset of the evaluation key terms specific to a single proof block. Because each input party should only provide its own input values and should not affect the values contributed by other parties, each proof block must be restricted to a subset of the wires. This is achieved by modifying Pinocchio's key generation such that, instead of a sampling a single value β, one such value, β_j, is sampled for each proof block j and the terms $\langle r_v \beta_j v_i + r_w \beta_j w_i + r_y \beta_j y_i \rangle_1$ are only included for wires indices i belonging to block j. That is, the jth block key is

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

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

A Trinocchio proof in the multi-client setting now consists of one block $\boldsymbol{Q}_i = (\langle V_i \rangle_1, \ldots, \langle Z_i \rangle_1)$ for each input and result party, one block $\boldsymbol{Q}_{\mathrm{mid}} = (\langle V_{\mathrm{mid}} \rangle_1, \ldots, \langle Z_{\mathrm{mid}} \rangle_1)$ 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 CheckBlock, cf. Algorithm 3. Compared to the Pinocchio verification key, our verification key contains "block verification keys" BV_i (i.e., elements $\langle \beta_j \rangle_1$ and $\langle \beta_j \rangle_2$) for each block instead of just $\langle \beta \rangle_1$ and $\langle \beta \rangle_2$. Apart from the relations inspected by CheckBlock, one other relation is needed to verify a Pinocchio proof: the divisibility check of Eq. (4) (Sect. 2.2). In the protocol, the algorithm that verifies this relation will be called CheckDiv. We denote the modified setup of the evaluation and verification keys by hybrid call MKeyGen.

5.2 Protocol Overview

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

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

- In the *input phase*, the input parties provide representations of their input on the bulletin board. These representations are later used as part of the proof to verify the computation results. They also serve to ensure that each input party provides its value independent of the other input values. The input parties then secret share their input values to the workers. The workers verify that the secret shared input values are consistent with their representations on the bulletin board, to prevent malicious input parties from providing a different value.
- The *computation phase* is very similar to the single-client variant of Trinocchio. In this phase the workers perform multi-party computation to carry out the actual computation and obtain secret shares of intermediate and result wire values. They then use these secret shared wire values to construct shares of the proof elements. These are then posted on the bulletin board, instead of being communicated directly to the result parties to ensure that all result parties receive a consistent result. In order to prevent these proof elements from revealing any information about the wire values, the zero-knowledge variant of the proof is used (Sect. 2.3).
- In the *result phase* the workers privately send the shares of the result values to the result parties. The result parties recombine the proof shares from the bulletin board and check whether the proof verifies. The result parties further check whether the recombined shares of the result are consistent with the information on the bulletin board. The result parties only accept the result received from the workers if both checks are satisfied.

5.3 Security of the Trinocchio Protocol

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

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

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

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

We give a proof of this theorem in the full version of the paper [SVdV15]. To prove secure function evaluation, we obtain privacy by simulating the multiparty computation of the proof with respect to the adversary without using honest inputs. To prove correct function evaluation, we run the protocol together with the adversary: if this gives a fake Pinocchio proof, then one of the underlying problems can be broken.

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

6 Performance

In this section, we show that our approach indeed adds privacy to verifiable computation with little overhead. We demonstrate this in a case study: we take the "MultiVar Poly" application from [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. In the full version of the paper we present a second case study in which we show that, using Trinocchio, the performance of "verification by validation" due to [dHSV16] can be considerably improved: in particular, we improve the client's performance by several orders of magnitude.

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 Sects. 4.3 and 5.3, this guarantees privacy against one passively corrupted worker (or, in the single-client case against θ actively corrupted workers when the multiparty computation protocol does not leak any information). We did not implement the multiple client scenario; this would add small overhead for the workers, with verification effort growing linearly in he number of input and result parties but remaining small and independent from the computation size. To simulate a realistic outsourcing scenario, we distribute computations between three Amazon EC2 "m3.medium" instances[3] around the world: one in Oregon, United States; one in Ireland; and one in Tokyo, Japan.

[3] Running Intel Xeon E5-2670 v2 Ivy Bridge with 4 GB SSD and 3.75 GiB RAM.

Multiparty computation requires secure and private channels: these are implemented using SSL.

6.1 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[4] 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 (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 Sect. 4, 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 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.

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

[4] Implementation available at http://meilof.home.fmf.nl/.

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)

	# 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

for its speed, so replacing VIFF with a different multiparty computation framework should considerably speed up the computation of f.

7 Discussion and Conclusion

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

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

As discussed, existing verifiable computation constructions in the single-worker setting [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, SV15, dHSV16] deliver privacy and correctness guarantees, but have much less efficient verification.

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

Our work is a first step towards privacy-preserving verifiable computation, and we see many promising directions for future work. Recent work in verifiable computation has extended the Pinocchio approach by making it easier to specify computations [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. 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. Another interesting direction is to investigate the possibility of practical universally composable [Can00b, CCL15] distributed verifiable computation; or to use the universal composability framework to obtain a more generic framework for combining multiparty computation with verifiable computation (even with only standalone guarantees).

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

References

[ACG+14] Ananth, P., Chandran, N., Goyal, V., Kanukurthi, B., Ostrovsky, R.: Achieving privacy in verifiable computation with multiple servers – without FHE and without pre-processing. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 149–166. Springer, Heidelberg (2014)

[AJCC15] Alderman, J., Janson, C., Cid, C., Crampton, J.: Access control in publicly verifiable outsourced computation. In: Proceedings of ASIACCS (2015)

[BDO14] Baum, C., Damgård, I., Orlandi, C.: Publicly auditable secure multi-party computation. In: Abdalla, M., De Prisco, R. (eds.) SCN 2014. LNCS, vol. 8642, pp. 175–196. Springer, Heidelberg (2014)

[BGW88] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Proceedings of STOC (1988)

[BSCG+13] Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part II. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013)

[Can00a] Canetti, R.: Security and composition of multi-party cryptographic protocols. J. Cryptology **13**(1), 143–202 (2000)

[Can00b] Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067 (2000)

[CCL15] Canetti, R., Cohen, A., Lindell, Y.: A simpler variant of universally composable security for standard multiparty computation. In: Gennaro, R., Robshaw, M. (eds.) CRYPTO 2015. LNCS, vol. 9216, pp. 3–22. Springer, Heidelberg (2015)

[CKKC13] Choi, S.G., Katz, J., Kumaresan, R., Cid, C.: Multi-client non-interactive verifiable computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 499–518. Springer, Heidelberg (2013)

[CLT14] Carter, H., Lever, C., Traynor, P.: Whitewash: outsourcing garbled circuit generation for mobile devices. In: Proceedings of ACSAC (2014)

[CTV15] Chiesa, A., Tromer, E., Virza, M.: Cluster computing in zero knowledge. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 371–403. Springer, Heidelberg (2015)

[DFK+06] Damgård, I.B., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876, pp. 285–304. Springer, Heidelberg (2006)

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

[dHSV16] de Hoogh, S., Schoenmakers, B., Veeningen, M.: Guaranteeing correctness in privacy-friendly outsourcing by certificate validation. In: Proceedings of AFRICACRYPT (2016)

[FGP14] Fiore, D., Gennaro, R., Pastro, V.: Efficiently verifiable computation on encrypted data. In: Proceedings of CCS (2014)

[GGP10] Gennaro, R., Gentry, C., Parno, B.: Non-interactive verifiable computing: outsourcing computation to untrusted workers. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 465–482. Springer, Heidelberg (2010)

[GGPR13] Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic span programs and succinct NIZKs without PCPs. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 626–645. Springer, Heidelberg (2013)

[GKL+15] Gordon, S.D., Katz, J., Liu, F.-H., Shi, E., Zhou, H.-S.: Multi-client verifiable computation with stronger security guarantees. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015, Part II. LNCS, vol. 9015, pp. 144–168. Springer, Heidelberg (2015)

[GKP+13] Goldwasser, S., Kalai, Y.T., Popa, R.A., Vaikuntanathan, V., Zeldovich, N.: Reusable garbled circuits and succinct functional encryption. In: Proceedings of STOC (2013)

[Gro10] Groth, J.: Short pairing-based non-interactive zero-knowledge arguments. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 321–340. Springer, Heidelberg (2010)

[GRR98] Gennaro, R., Rabin, M.O., Rabin, T.: Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In: Proceedings of PODC (1998)

[JNO14] Jakobsen, T.P., Nielsen, J.B., Orlandi, C.: A framework for outsourcing of secure computation. In: Proceedings of CCSW (2014)

[KMR12] Kamara, S., Mohassel, P., Riva, B.: Salus: a system for server-aided secure function evaluation. In: Proceedings of CCS (2012)

[MF06] Mohassel, P., Franklin, M.K.: Efficiency tradeoffs for malicious two-party computation. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 458–473. Springer, Heidelberg (2006)

[Mit13] Mitsunari, S.: A fast implementation of the optimal ate pairing over BN curve on Intel Haswell processor. Cryptology ePrint Archive, Report 2013/362 (2013)

[PHGR13] Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: Proceedings of S&P (2013)

[PTK13] Peter, A., Tews, E., Katzenbeisser, S.: Efficiently outsourcing multiparty computation under multiple keys. IEEE Trans. Inf. Forensics Secur. **8**(12), 2046–2058 (2013)

[ST06] Schoenmakers, B., Tuyls, P.: Efficient binary conversion for Paillier encrypted values. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 522–537. Springer, Heidelberg (2006)

[SV15] Schoenmakers, B., Veeningen, M.: Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In: Liu, S., et al. (eds.) ACNS 2015. LNCS, vol. 9092, pp. 3–22. Springer, Heidelberg (2015). doi:10.1007/978-3-319-28166-7_1

[SVdV15] Schoenmakers, B., Veeningen, M., de Vreede, N.: Trinocchio: privacy-friendly outsourcing by distributed verifiable computation. Cryptology ePrint Archive, Report 2015/480 (2015)