Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation

Meilof Veeningen(✉)

Philips Research, Eindhoven, The Netherlands
meilof.veeningen@philips.com

Abstract. Pinocchio is a practical zk-SNARK that allows a prover to perform cryptographically verifiable computations with verification effort potentially less than performing the computation itself. A recent proposal showed how to make Pinocchio adaptive (or "hash-and-prove"), i.e., to enable proofs with respect to computation-independent commitments. This enables computations to be chosen after the commitments have been produced, and for data to be shared between different computations in a flexible way. Unfortunately, this proposal is not zero-knowledge. In particular, it cannot be combined with Trinocchio, a system in which Pinocchio is outsourced to three workers that do not learn the inputs thanks to multi-party computation (MPC). In this paper, we show how to make Pinocchio adaptive in a zero-knowledge way; apply this to make Trinocchio work on computation-independent commitments; present tooling to easily program flexible verifiable computations (with or without MPC); and use it to build a prototype in a medical research case study.

1 Introduction

Recent advances in SNARKs (Succinct Non-interactive ARguments of Knowledge) are making it more and more feasible to outsource computations to the cloud while obtaining cryptographic guarantees about the correctness of their outputs. In particular, the Pinocchio system [8,11] achieved for the first time for a practical computation a verification time of a computation proof that was actually faster than performing the computation itself.

In Pinocchio, proofs are verified with respect to plaintext inputs and outputs of the verifier; but in many cases, it is useful to have computation proofs that also refer to committed data, e.g., provided by a third party. Ideally, such proofs should be *adaptive*, i.e., multiple different computations can be performed on the same commitment, that are chosen after the data has been committed to; and *zero-knowledge*, i.e., the commitments and proofs should reveal no information about the committed data. This latter property allows proofs on sensitive data, and it allows extensions like Trinocchio [13] that additionally hide this sensitive data from provers by multi-party computation.

Although several approaches are known from the literature, no really satisfactory practical adaptive zk-SNARK exists. The recent "hash first" proposal [7]

M. Joye and A. Nitaj (Eds.): AFRICACRYPT 2017, LNCS 10239, pp. 21–39, 2017.
DOI: 10.1007/978-3-319-57339-7_2

shows how to make Pinocchio adaptive at low overhead, but is unfortunately not zero-knowledge. On the other hand, Pinocchio's successor Geppetto [3] is zero-knowledge but not adaptive: multiple computations can be performed on the same data but they need to be known before committing. The asymptotically best known SNARKS combining the two properties have $\Theta(n \log n)$ non-cryptographic and $\Theta(n)$ cryptographic work for the prover, a $\Theta(n)$-sized CRS, and constant-time verification (where n is the size of the computation), but with a large practical overhead: [10] because it relies on the impractical subset-sum language; other constructions (e.g., [3,7]) because they rely on including hash evaluation in the computation[1]. Finally, [1] enables Pinocchio proofs on authenticated data with prover complexity as above, but verification time is linear in the number of committed inputs.

In this work, we give a new Pinocchio-based adaptive zk-SNARK that solves the above problems. We match the best asymptotic performance (i.e., $\Theta(n \log n)$ non-cryptographic work and $\Theta(n)$ cryptographic work for the prover; a $\Theta(n)$-size CRS and constant-time verification); but obtain the first practical solution by adding only minor overhead to "plain" Pinocchio (instead of relying on expensive approaches such as subset-sum or bootstrapping).

As additional contributions, we apply our zk-SNARK in the Trinocchio setting, and present tooling to easily perform verifiable computations. Trinocchio [13] achieves privacy-preserving outsourcing to untrusted workers by combining the privacy guarantees of multi-party computation with the correctness guarantees of the Pinocchio zk-SNARK. With our adaptive zk-SNARK, computation can be chosen *after* the inputs were provided and more complex functionalities can be achieved by using the output of one computation as input of another. We also improve the generality of [13] by proving security for *any* suitable MPC protocol and adaptive zk-SNARK. Our tooling consists of a Python frontend and a C++ backend. The frontend allows easy programming of verifiable computations (with libraries for zero testing, oblivious indexing and fixed-point computations), and execution either directly (for normal outsourcing scenarios) or with MPC (for privacy-preserving outsourcing). The backend provides key generation, proving, and verification functionality for both scenarios.

2 Preliminaries

2.1 Algebraic Tools, Notation, and Complexity Assumptions

Our constructions aim to prove correctness of computations over a prime order field $\mathbb{F} = \mathbb{F}_p$. We make use of pairings, i.e., groups $(\mathbb{G}, \mathbb{G}', \mathbb{G}_T)$ of order p and an efficient bilinear map $e : \mathbb{G} \times \mathbb{G}' \to \mathbb{G}_T$, where for any generators $g_1 \in \mathbb{G}, g_2 \in \mathbb{G}'$, $e(g_1, g_2) \neq 1$ and $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$. Throughout, we will make use of polynomial evaluations in a secret point $s \in \mathbb{F}$. For $f \in \mathbb{F}[x]$, write $\langle f \rangle_1$ for $f(s) \cdot g_1$ and $\langle f \rangle_2$ for $f(s) \cdot g_2$.

[1] In practice, computing the hash is complex itself. It can be avoided with bootstrapping [10], giving slightly worse asymptotics and again a large practical overhead.

Let $(\mathbb{G}, \mathbb{G}', \mathbb{G}_T, e) \leftarrow \mathcal{G}(1^\kappa)$ denote parameter generation in this setting. We require the following assumptions from [4] (that generalise those from [11] to asymmetric pairings):

Definition 1. *The* q-power Diffie Hellman (q-PDH) assumption *holds for* $\mathcal{G}$ *if, for any NUPPT adversary* $\mathcal{A}$: $\Pr[(\mathbb{G}, \mathbb{G}', \mathbb{G}_T, e) \leftarrow \mathcal{G}(1^\kappa);\ g \in_R \mathbb{G}^*; g' \in_R \mathbb{G}'^*;\ s \in_R \mathbb{Z}_p^*; y \leftarrow \mathcal{A}(\mathbb{G}, \mathbb{G}', \mathbb{G}_T, e, \{g^{s^i}, g'^{s^i}\}_{i=1,\ldots,q,q+2,\ldots,2q}) : y = g^{s^{q+1}}] \approx_\kappa 0$.

Definition 2. *The* q-power knowledge of exponent (q-PKE) assumption *holds for* $\mathcal{G}$ *and a class* $\mathcal{Z}$ *of auxiliary input generators if, for every NUPPT auxiliary input generator* $Z \in \mathcal{Z}$ *and any NUPPT adversary* $\mathcal{A}$ *there exists a NUPPT extractor* $\mathcal{E}_\mathcal{A}$ *such that:* $\Pr[crs := (\mathbb{G}, \mathbb{G}', \mathbb{G}_T, e) \leftarrow \mathcal{G}(1^\kappa);\ g \in_R \mathbb{G}^*;\ s \in_R \mathbb{Z}_p^*;\ z \leftarrow Z(crs, g, \ldots, g^{s^q}); g' \in_R \mathbb{G}'^*; (c, c' || a_0, \ldots, a_q) \leftarrow (\mathcal{A}||\mathcal{E}_\mathcal{A})(crs, \{g^{s^i}, g'^{s^i}\}_{i=0,\ldots,q}, z) : e(c, g') = e(g, c') \ \wedge\ c \neq \prod_{i=0}^q g_1^{a_i s^i}] \approx_\kappa 0$.

Here, $(a||b) \leftarrow (\mathcal{A}||\mathcal{E}_\mathcal{A})(c)$ denotes running both algorithms on the same inputs and random tape, and assigning their results to a respectively b. For certain auxiliary input generators, the q-PKE assumption does not hold, so we have to conjecture that our auxiliary input generators are "benign", cf. [4].

Definition 3. *The* q-target group strong Diffie Hellman (q-SDH) assumption *holds for* $\mathcal{G}$ *if, for any NUPPT adversary* $\mathcal{A}$, $\Pr[crs := (\mathbb{G}, \mathbb{G}', \mathbb{G}_T, e) \leftarrow \mathcal{G}(1^\kappa); g \in_R \mathbb{G}^*; g' \in_R \mathbb{G}'^*; s \in_R \mathbb{Z}_p(r, Y) \leftarrow \mathcal{A}(crs, \{g^{s^i}, g'^{s^i}\}_{i=0,\ldots,q}) : r \in \mathbb{Z}_p \setminus \{s\} \wedge Y = e(g, g')^{\frac{1}{s-r}}] \approx_\kappa 0$.

2.2 Adaptive zk-SNARKs in the CRS Model

We now define adaptive zk-SNARKs as in [10] with minor modifications. We first define extractable trapdoor commitment families. This is a straightforward generalisation of an extractable trapdoor commitment scheme [10] that explicitly captures multiple commitment keys generated from the same CRS:

Definition 4. *Let* $(\mathsf{G0}, \mathsf{Gc}, \mathsf{C})$ *be a scheme where* $(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{G0}(1^\kappa)$ *outputs a system-wide CRS and a trapdoor;* $(\mathsf{ck}, \mathsf{ctd}) \leftarrow \mathsf{Gc}(\mathsf{crs})$ *outputs a commitment key and a trapdoor; and* $c \leftarrow \mathsf{ck}(m; r)$ *outputs a commitment with the given key. Such a scheme is called an* extractable trapdoor commitment family *if:*

- *(Computationally binding) For every NUPPT* $\mathcal{A}$, $\Pr[(\mathsf{crs}, \cdot) \leftarrow \mathsf{G0}(1^\kappa); (\mathsf{ck}, \cdot) \leftarrow \mathsf{Gc}(\mathsf{crs}); (v; r; v'; r') \leftarrow \mathcal{A}(\mathsf{crs}; \mathsf{ck}) : \mathsf{C}_{\mathsf{ck}}(v; r) = \mathsf{C}_{\mathsf{ck}}(v'; r')] \approx 0$.
- *(Perfectly hiding) Letting* $(\mathsf{crs}, \cdot) \leftarrow \mathsf{G0}(1^\kappa)$*;* $(\mathsf{ck}, \cdot) \leftarrow \mathsf{Gc}(\mathsf{crs})$*, for all* v, v'*,* $\mathsf{C}_{\mathsf{ck}}(v; r)$ *and* $\mathsf{C}_{\mathsf{ck}}(v'; r')$ *are identically distributed given random* r, r'
- *(Trapdoor) There exists a NUPPT algorithm* $\mathcal{T}$ *such that if* $(\mathsf{crs}, \mathsf{td}) \leftarrow \mathsf{G0}(1^\kappa)$*;* $(\mathsf{ck}; \mathsf{ctd}) \leftarrow \mathsf{Gc}(\mathsf{crs})$*;* $(u; t) \leftarrow \mathcal{T}(\mathsf{crs}; \mathsf{td}; \mathsf{ck}; \mathsf{ck})$*;* $r \leftarrow \mathcal{T}(t; u; v)$*, then* u *is distributed identically to real commitments and* $\mathsf{C}_{\mathsf{ck}}(v; r) = u$.
- *(Extractable) For every NUPPT committer* $\mathcal{A}$*, there exists a NUPPT extractor* $\mathcal{E}_\mathcal{A}$ *such that* $\Pr[(\mathsf{crs}; \cdot) \leftarrow \mathsf{G0}(1^\kappa); (\mathsf{ck}; \cdot) \leftarrow \mathsf{Gc}(\mathsf{crs}); (u||v; r) \leftarrow (\mathcal{A}||\mathcal{E}_\mathcal{A})(\mathsf{crs}; \mathsf{ck}) : u \in \mathit{Range}(\mathsf{C}_{\mathsf{ck}}) \ \wedge\ u \neq \mathsf{C}_{\mathsf{ck}}(v; r)] \approx 0$.

Given relation $\mathcal{R}$ and commitment keys $\mathsf{ck}_1, \ldots, \mathsf{ck}_n$ from the same commitment family, define: $\mathcal{R}_{\mathsf{ck}_1,\ldots,\mathsf{ck}_n} := \{(\boldsymbol{u}; \boldsymbol{v}, \boldsymbol{r}, \boldsymbol{w}) : \boldsymbol{u_i} = \mathsf{C}_{\mathsf{ck}_i}(\boldsymbol{v_i}; \boldsymbol{r_i}) \wedge (\boldsymbol{v}; \boldsymbol{w}) \in \mathcal{R}\}$. Intuitively, an adaptive zk-SNARK is a zk-SNARK for relation $\mathcal{R}_{\mathsf{ck}_1,\ldots,\mathsf{ck}_n}$.

Definition 5. *An adaptive zk-SNARK for extractable trapdoor commitment family* $(\mathsf{G0}, \mathsf{Gc}, \mathsf{C})$ *and relation* $\mathcal{R}$ *is a scheme* $(\mathsf{G}, \mathsf{P}, \mathsf{V})$ *where*[2]*:*

- $(\mathsf{crsp}; \mathsf{crsv}; \mathsf{tdp}) \leftarrow \mathsf{G}(\mathsf{crs}; \{\mathsf{ck}_i\})$*, given a CRS and commitment keys, outputs evaluation and verification keys, and a trapdoor;*
- $\pi \leftarrow \mathsf{P}(\mathsf{crs}; \{\mathsf{ck}_i\}; \mathsf{crsp}; \boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w})$*, given a CRS; commitment keys; an evaluation key; commitments; openings; and a witness, outputs a proof;*
- $0/1 \leftarrow \mathsf{V}(\mathsf{crs}; \{\mathsf{ck}_i\}; \mathsf{crsv}; \boldsymbol{u}; \pi)$*, given a CRS; commitment keys; a verification key; commitments; and a proof, verifies the proof,*

satisfying the following properties (let $\mathsf{setup} := (\mathsf{crs}; \mathsf{td}) \leftarrow \mathsf{G0}(1^\kappa); \forall i : (\mathsf{ck}_i; \mathsf{ctd}_i) \leftarrow \mathsf{Gc}(\mathsf{crs}); (\mathsf{crsp}; \mathsf{crsv}; \mathsf{tdp}) \leftarrow \mathsf{G}(\mathsf{crs}; \{\mathsf{ck}_i\})$*):*

- Perfect completeness *("proofs normally verify")* $\Pr[\mathsf{setup}; (\boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w}) \leftarrow \mathcal{R}_{\{\mathsf{ck}_i\}} : \mathsf{V}(\mathsf{crs}; \{\mathsf{ck}_i\}; \mathsf{crsv}; \boldsymbol{u}; \mathsf{P}(\mathsf{crs}; \{\mathsf{ck}_i\}; \mathsf{crsp}; \boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w})) = 1] = 1$.
- Argument of knowledge *("the commitment openings and a valid witness can be extracted from an adversary producing a proof"): for every NUPPT* $\mathcal{A}$ *there exists NUPPT extractor* $\mathcal{E}_\mathcal{A}$ *such that, for every auxiliary information* $\mathsf{aux} \in \{0,1\}^{poly(\kappa)}$*:* $\Pr[\mathsf{setup}; (\boldsymbol{u}; \pi || \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w}) \leftarrow (\mathcal{A} || \mathcal{E}_\mathcal{A})(\mathsf{crs}; \{\mathsf{ck}_i\}; \mathsf{crsp}; \mathsf{aux} || \ldots; \mathsf{td}; \mathsf{ctd}_1; \ldots; \mathsf{ctd}_n; \mathsf{tdp}) : (\boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w}) \notin \mathcal{R}_{\{\mathsf{ck}_i\}} \wedge \mathsf{V}(\mathsf{crs}; \{\mathsf{ck}_i\}; \mathsf{crsv}; \boldsymbol{u}; \pi) = 1] \approx_\kappa 0$*. Here,* $(\mathcal{A} || \mathcal{E}_\mathcal{A})(\cdot || \ldots; \cdot')$ *is parallel execution with extra input* $\cdot'$ *for* $\mathcal{E}_\mathcal{A}$*.*
- Perfectly composable zero knowledge *("proofs can be simulated using the commitments and trapdoor"): there exists a PPT simulator* $\mathcal{S}$ *such that, for all stateful NUPPT adversaries* $\mathcal{A}$*,* $\Pr[\mathsf{setup}; (\boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w}) \leftarrow \mathcal{A}(\mathsf{crs}, \{\mathsf{ck}_i\}, \mathsf{crsp}); \pi \leftarrow \mathsf{P}(\mathsf{crs}, \{\mathsf{ck}_i\}, \mathsf{crsp}; \boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w}) : (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{r}, \boldsymbol{w}) \in \mathcal{R}_{\{\mathsf{ck}_i\}} \wedge \mathcal{A}(\pi) = 1] = \Pr[\mathsf{setup}; (\boldsymbol{u}; \boldsymbol{v}; \boldsymbol{r}; \boldsymbol{w}) \leftarrow \mathcal{A}(\mathsf{crs}, \{\mathsf{ck}_i\}, \mathsf{crsp}); \pi \leftarrow \mathcal{S}(\mathsf{crs}, \{\mathsf{ck}_i\}, \mathsf{crsp}; \boldsymbol{u}; \mathsf{td}, \{\mathsf{ctd}_i\}, \mathsf{tdp}) : (\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{r}, \boldsymbol{w}) \in \mathcal{R}_{\{\mathsf{ck}_i\}} \wedge \mathcal{A}(\pi) = 1]$.

We base our definitions on [10] because it is closest to what we want to achieve. Unlike in [3], we do not guarantee security when relation $\mathcal{R}$ is chosen adaptively based on the commitment keys; this is left as future work.

2.3 The Pinocchio zk-SNARK Construction from [11]

QAPs. Pinocchio models computations as *quadratic arithmetic programs* (QAPs) [8]. A QAP over a field $\mathbb{F}$ is a triple $(\mathbf{V}, \mathbf{W}, \mathbf{Y}) \in (\mathbb{F}^{d \times k})^3$, where d is called the *degree* of the QAP and k is called the *size*. A vector $\boldsymbol{x} \in \mathbb{F}^k$ is said to be a

[2] We differ from [10] in three minor ways: (1) we generalise from commitment schemes to families because we need this in Adaptive Trinocchio; (2) we allow witnesses that are not committed to separately, giving a slight efficiency improvement; (3) the extractor has access to the trapdoor, as needed when using Pinocchio [8].

solution to the QAP if $(\mathbf{V}\cdot\boldsymbol{x})\times(\mathbf{W}\cdot\boldsymbol{x})=\mathbf{Y}\cdot\boldsymbol{x}$, where $\times$ denotes the pairwise product and $\cdot$ denotes normal matrix-vector multiplication. A QAP Q is said to *compute* function $f:\mathbb{F}^i\to\mathbb{F}^j$ if $\boldsymbol{b}=f(\boldsymbol{a})$ and only if there exists a *witness* $\boldsymbol{w}$ such that $(\boldsymbol{a};\boldsymbol{b};\boldsymbol{w})$ is a solution to Q. For example, consider the QAP

$$\mathbf{V}=\begin{pmatrix}1&1&0&0\\1&1&0&0\end{pmatrix},\quad \mathbf{W}=\begin{pmatrix}1&1&0&0\\0&0&0&1\end{pmatrix},\quad \mathbf{Y}=\begin{pmatrix}0&0&0&1\\0&0&1&0\end{pmatrix}.$$

Intuitively, the first row of this QAP represents equation $(x_1+x_2)\cdot(x_1+x_2)=x_4$ in variables (x_1,x_2,x_3,x_4) whereas the second row represents equation $(x_1+x_2)\cdot x_4=x_3$. Note that $x_3=(x_1+x_2)^3$ if and only if there exists x_4 satisfying the two equations, so this QAP computes function $f:(x_1,x_2)\mapsto x_3$.[3]

Fixing d distinct, public $\omega_1,\ldots,\omega_d\in\mathbb{F}$, then a QAP can equivalently be described by a collection of interpolating polynomials in these points. Namely, let $\{v_i(x)\}$ be the unique polynomials of degree $<d$ such that $v_i(\omega_j)=\mathbf{V}_{j,i}$, and similarly for $\{w_i(x)\}$, $\{y_i(x)\}$. Then $\{v_i(x),w_i(x),y_i(x)\}$ is an equivalent description of the QAP. Defining $t(x)=(x-\omega_1)\cdot\ldots\cdot(x-\omega_d)\in\mathbb{F}[x]$, note that $\boldsymbol{x}_1,\ldots,\boldsymbol{x}_n$ is a solution to Q if and only if, for all j, $(\sum_i \boldsymbol{x}_i\cdot v_i(\omega_j))\cdot(\sum_i \boldsymbol{x}_i\cdot w_i(\omega_j))=(\sum_i \boldsymbol{x}_i\cdot y_i(\omega_j))$, or equivalently, if $t(x)$ divides $p(x):=(\sum_i \boldsymbol{x}_i\cdot v_i(x))\cdot(\sum_i \boldsymbol{x}_i\cdot w_i(x))-(\sum_i \boldsymbol{x}_i\cdot y_i(x))\in\mathbb{F}[x]$.

Security Guarantees. Pinocchio is a zk-SNARK, which is essentially the same as an adaptive zk-SNARK except proving and verifying are with respect to plaintext values instead of commitments. In Pinocchio, relation $\mathcal{R}$ is that, for given $\boldsymbol{v}$, there exists witness $\boldsymbol{w}$ such that $(\boldsymbol{v};\boldsymbol{w})$ is a solution to a given QAP Q. Pinocchio replies on a pairing secure under the $(4d+4)$-PDH, d-PKE and $(8d+8)$-SDH assumptions discussed above, where d is the degree of the QAP.

Construction. Fix random, secret $s,\alpha_v,\alpha_w,\alpha_y,\beta,r_v,r_w,r_y(x):=r_vr_w$. The central idea of Pinocchio is to prove satisfaction of all QAP equations using evaluations of the interpolating polynomials in a secret point. Namely, the prover computes quotient polynomial $h=p/t$ and basically provides evaluations "in the exponent" of h, $\sum_i \boldsymbol{x}_i\cdot v_i$, $\sum_i \boldsymbol{x}_i\cdot w_i$, $\sum_i \boldsymbol{x}_i\cdot y_i$ in the point s that is unknown to him, that can then be verified using the pairing. Precisely, the prover algorithm, given solution $\boldsymbol{x}=(\boldsymbol{v};\boldsymbol{w})$ to the QAP, generates random $\delta_v,\delta_w,\delta_y$; computes coefficients $\boldsymbol{h}$ of the polynomial $(\sum_i \boldsymbol{x}_i\cdot v_i(x)+\delta_v t(x))\cdot(\sum_i \boldsymbol{x}_i\cdot w_i(x)+\delta_w t(x))-(\sum_i \boldsymbol{x}_i\cdot y_i(x)+\delta_y t(x))/t(x)$ (with δ terms added to make the proof zero-knowledge), and outputs (all $\sum_i$ over witness indices $|\boldsymbol{v}|+1,\ldots,|\boldsymbol{x}|$; recall that for polynomial f, $\langle f\rangle_1:=f(s)\cdot g_1$ and $\langle f\rangle_2:=f(s)\cdot g_2$):

[3] In Pinocchio, the linear terms corresponding to $\mathbf{V}$, $\mathbf{W}$, $\mathbf{Y}$ can also contain constant values. This is achieved by assigning special meaning to a "constant" wire with value 1. We do not formalise this separately, instead leaving it up to the user to include a special variable and an equation $x_i\cdot x_i=x_i$ that forces this variable to be one.

$$\langle V\rangle_1 = \sum_i \boldsymbol{x}_i\langle r_v v_i\rangle_1 + \delta_v\langle r_v t\rangle_1, \langle \alpha_v V\rangle_2 = \sum_i \boldsymbol{x}_i\langle \alpha_v r_v v_i\rangle_2 + \delta_v\langle \alpha_v r_v t\rangle_2,$$
$$\langle W\rangle_2 = \sum_i \boldsymbol{x}_i\langle r_w w_i\rangle_2 + \delta_w\langle r_w t\rangle_2, \langle \alpha_w W\rangle_1 = \sum_i \boldsymbol{x}_i\langle \alpha_w r_w w_i\rangle_1 + \delta_w\langle \alpha_w r_w t\rangle_1,$$
$$\langle Y\rangle_1 = \sum_i \boldsymbol{x}_i\langle r_y y_i\rangle_1 + \delta_y\langle r_y t\rangle_1, \langle \alpha_y Y\rangle_2 = \sum_i \boldsymbol{x}_i\langle \alpha_y r_y y_i\rangle_2 + \delta_y\langle \alpha_y r_y t\rangle_2.$$
$$\langle Z\rangle_1 = \sum_i \boldsymbol{x}_i\langle r_v\beta v_i + r_w\beta w_i + r_y\beta y_i\rangle_1 + \delta_v\langle r_v\beta t\rangle_1 + \delta_w\langle r_w\beta t\rangle_1 + \delta_y\langle r_y\beta t\rangle_1,$$
$$\langle H\rangle_1 = \sum_{j=0}^{d} \boldsymbol{h}_j\langle x^j\rangle_1.$$

The evaluation key consists of all $\langle\cdot\rangle_1$, $\langle\cdot\rangle_2$ items used in the formulas above.[4]

The verification algorithm, given statement $\boldsymbol{v}$, extends $\langle V\rangle_1$, $\langle W\rangle_1$, $\langle Y\rangle_1$ to include also the input/output wires ($\sum_i$ over I/O wire indices $1,\ldots,|\boldsymbol{v}|$): $\langle V^+\rangle_1 = \langle V\rangle_1 + \sum_i \boldsymbol{x}_i\langle r_v v_i\rangle_1$, $\langle W^+\rangle_2 = \langle W\rangle_2 + \sum_i \boldsymbol{x}_i\langle r_w w_i\rangle_2$, $\langle Y^+\rangle_1 = \langle Y\rangle_1 + \sum_i \boldsymbol{x}_i\langle r_y y_i\rangle_1$. Then, it checks (the verification key are the needed $\langle\cdot\rangle_1$, $\langle\cdot\rangle_2$ items):

$$e(\langle V\rangle_1, \langle\alpha_v\rangle_2) = e(\langle 1\rangle_1, \langle\alpha_v V\rangle_2); \quad (\mathbf{V})$$
$$e(\langle\alpha_w\rangle_1, \langle W\rangle_2) = e(\langle\alpha_w W\rangle_1, \langle 1\rangle_2); \quad (\mathbf{W})$$
$$e(\langle Y\rangle_1, \langle\alpha_y\rangle_2) = e(\langle 1\rangle_1, \langle\alpha_y Y\rangle_2); \quad (\mathbf{Y})$$
$$e(\langle V\rangle_1 + \langle Y\rangle_1, \langle\beta\rangle_2)\cdot e(\langle\beta\rangle_1, \langle W\rangle_2) = e(\langle Z\rangle_1, \langle 1\rangle_2); \quad (\mathbf{Z})$$
$$e(\langle V^+\rangle_1, \langle W^+\rangle_2)\cdot e(\langle Y^+\rangle_1, \langle 1\rangle_2)^{-1} = e(\langle H\rangle_1, \langle r_y t\rangle_2). \quad (\mathbf{H})$$

At a high level, checks (**V**), (**W**), (**Y**) guarantee that the proof is a proof of knowledge of the witness $\boldsymbol{w}$; check (**Z**) guarantees that the same witness $\boldsymbol{w}$ was used for $\langle V\rangle_1, \langle W\rangle_2, \langle Y\rangle_1$; and check (**Z**) guarantees that indeed, $p(x) = h(x)\cdot t(x)$ holds, which implies a solution to the QAP.

3 Adaptive zk-SNARKs Based on Pinocchio

This section presents the central contribution of this paper: an adaptive zk-SNARK based on Pinocchio. We obtain our Pinocchio-based adaptive zk-SNARK by generalising the role of the $\langle Z\rangle_1$ element of the Pinocchio proof. Recall that in Pinocchio, proof elements $\langle V\rangle_1$, $\langle W\rangle_1$, and $\langle Y\rangle_1$ are essentially weighted sums $\sum_j \boldsymbol{x}_j\langle v_j\rangle_1$, $\sum_j \boldsymbol{x}_j\langle w_j\rangle_2$, $\sum_j \boldsymbol{x}_j\langle y_j\rangle_1$ over elements $\langle v_j\rangle_1$, $\langle w_j\rangle_2$, $\langle y_j\rangle_1$ from the CRS, with the weights given by the witness part of the QAP's solution vector $\boldsymbol{x}$. The $\langle Z\rangle_1$ element ensures that these weighted sums consistently use the same witness. This is done by forcing the prover to come up essentially with $\beta\cdot(\langle V\rangle_1 + \langle W\rangle_2 + \langle Y\rangle_1)$ given only elements $\langle\beta\cdot(v_j + w_j + y_j)\rangle_1$ in which v_j, w_j, and y_j occur together. The essential idea is of our construction is to use the $\langle Z\rangle_1$ element also to ensure consistency to external commitments.

In more detail, in earlier works [3,13], it was noted that the Pinocchio $\langle V\rangle_1, \langle W\rangle_2, \langle Y\rangle_1$ elements can be divided into multiple "blocks" ($\langle V_i\rangle_1$, $\langle W_i\rangle_2$, $\langle Y_i\rangle_1, \langle Z_i\rangle_1$). Each block contains the values of a number of variables of the

[4] We use $\langle\alpha_v V\rangle_2$ etc. instead of $\langle\alpha_v V\rangle_1$ from [13], so that we can rely on the asymmetric q-PKE assumption from [4] (which [13] did not spell out).

Extractable Trapdoor Commitment Scheme Family $(\mathsf{G0}^1, \mathsf{Gc}^1, \mathsf{C}^1)$**:**

- $\mathsf{G0}^1$: Fix $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_3$ and random s. Return $\mathsf{crs} = (\{\langle x^i\rangle_1, \langle x^i\rangle_2\}_{i=0,\ldots,d})$, $\mathsf{td} = s$.
- Gc^1: Pick random α. Return $\mathsf{ck} = (\langle 1\rangle_1, \langle\alpha\rangle_2, \langle x\rangle_1, \langle\alpha x\rangle_2, \ldots, \langle x^d\rangle_1, \langle\alpha x^d\rangle_2)$
- C^1: Return $(r\langle 1\rangle_1 + \boldsymbol{v}_1\langle x\rangle_1 + \boldsymbol{v}_2\langle x^2\rangle_1 + \ldots, r\langle\alpha\rangle_2 + \boldsymbol{v}_1\langle\alpha x\rangle_2 + \boldsymbol{v}_2\langle\alpha x^2\rangle_2 + \ldots)$

Key generation G^1**:** Fix a QAP of degree at most d, and let $v_j(x), w_j(x), y_j(x)$ be as in Pinocchio. Fix random, secret $\alpha_v, \alpha_w, \alpha_y, \beta, r_v, r_w$. Let $r_y = r_v r_w$. Let $z_j(x) = x^j + r_v v_j + r_w w_j + r_y y_j$ if $j \leq W$ and $z_j(x) = r_v v_j + r_w w_j + r_y y_j$ otherwise. Evaluation key $(i = 1, \ldots, n,\ j = 1, \ldots, d)$:

$$\langle x^j\rangle_1, \langle r_v v_j\rangle_1, \langle r_v t\rangle_1, \langle\alpha_v r_v v_j\rangle_2, \langle\alpha_v r_v t\rangle_2 \langle r_w w_j\rangle_1, \langle r_w t\rangle_1, \langle\alpha_w r_w w_j\rangle_1, \langle\alpha_w r_w t\rangle_1 \langle r_y y_j\rangle_1,$$
$$\langle r_y t\rangle_1, \langle\alpha_y r_y y_j\rangle_2, \langle\alpha_y r_y t\rangle_2 \langle\beta_i z_{(i-1)d+j}\rangle_1, \langle\beta_i z_{nd+j}\rangle_1, \langle\beta_i\rangle_1, \langle\beta_i r_v t\rangle_1, \langle\beta_i r_w t\rangle_1, \langle\beta_i r_y t\rangle_1$$

Verification key $(i = 1, \ldots, n)$: $(\langle\alpha_v\rangle_2, \langle\alpha_w\rangle_1, \langle\alpha_y\rangle_2, \langle\beta_i\rangle_2, \langle\beta_i\rangle_1, \langle r_y t\rangle_2)$.

Proof generation P^1**:** Let $\boldsymbol{u}_i = \mathsf{C}^1_{\mathsf{ck}_i}(\boldsymbol{v_i}; r_i)$, and let $\boldsymbol{w}$ be the witness such that $(\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}; \boldsymbol{w})$ is a solution to the QAP. Generate random $\delta_{v,i}, \delta_{w,i}, \delta_{y,i}$. Compute $\boldsymbol{h}$ as the coefficients of polynomial $((\sum_j \boldsymbol{x}_j \cdot v_j(x) + \delta_v \cdot t(x)) \cdot (\sum_j \boldsymbol{x}_j \cdot w_j(x) + \delta_w \cdot t(x)) - (\sum_j \boldsymbol{x}_j \cdot y_j(x) + \delta_y \cdot t(x)))/t(x)$. Return $(i = 1, \ldots, n$; $[\cdot]$ means only if $i = 1)$:

$$\langle V_i\rangle_1 = \textstyle\sum_{j=1}^d \boldsymbol{v_{i,j}} \langle r_v v_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \boldsymbol{w}_j \langle r_v v_{nd+j}\rangle_1\right] + \delta_{v,i}\langle r_v t\rangle_1, \langle\alpha_v V_i\rangle_2 = \ldots$$
$$\langle W_i\rangle_1 = \textstyle\sum_{j=1}^d \boldsymbol{v_{i,j}} \langle r_w w_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \boldsymbol{w}_j \langle r_w w_{nd+j}\rangle_1\right] + \delta_{w,i}\langle r_w t\rangle_1, \langle\alpha_w W_i\rangle_1 = .$$
$$\langle Y_i\rangle_1 = \textstyle\sum_{j=1}^d \boldsymbol{v_{i,j}} \langle r_y y_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \boldsymbol{w}_j \langle r_y y_{nd+j}\rangle_1\right] + \delta_{y,i}\langle r_y t\rangle_1, \langle\alpha_y Y_i\rangle_2 = \ldots$$
$$\langle Z_i\rangle_1 = \textstyle\sum_{j=1}^d \boldsymbol{v_{i,j}} \langle\beta_i z_{(i-1)d+j}\rangle_1 \left[+ \sum_{j=1}^N \boldsymbol{w}_j \langle\beta_i z_{nd+j}\rangle_1\right] + r_i\langle\beta_i\rangle_1 + \delta_{v,i}\langle\beta_i r_v t\rangle_1$$
$$\langle H\rangle_1 = \textstyle\sum_j \boldsymbol{h}_j \langle x^j\rangle_1. \qquad + \delta_{w,i}\langle\beta_i r_w t\rangle_1 + \delta_{y,i}\langle\beta_i r_y t\rangle_1$$

Proof verification V^1**:** Letting $\mathsf{ck}_i = (\ldots, \langle\alpha_i\rangle_2)$, $\boldsymbol{u}_i = (\langle C_i\rangle_1, \langle\alpha_i C_i\rangle_2)$, check that:

$$e(\langle C_i\rangle_1, \langle\alpha_i\rangle_2) = e(\langle 1\rangle_1, \langle\alpha_i C_i\rangle_2); \quad e(\langle V_i\rangle_1, \langle\alpha_v\rangle_2) = e(\langle\alpha_v V_i\rangle_1, \langle 1\rangle_2); \qquad \textbf{(C,V)}$$
$$e(\langle\alpha_w\rangle_1, \langle W_i\rangle_2) = e(\langle 1\rangle_1, \langle\alpha_w W_i\rangle_2); \quad e(\langle Y_i\rangle_1, \langle\alpha_y\rangle_2) = e(\langle 1\rangle_1, \langle\alpha_y Y_i\rangle_2); \qquad \textbf{(W,Y)}$$
$$e(\langle V_i\rangle_1 + \langle Y_i\rangle_1 + \langle C_i\rangle_1, \langle\beta_i\rangle_2) \cdot e(\langle\beta_i\rangle_1, \langle W_i\rangle_2) = e(\langle Z_i\rangle_1, \langle 1\rangle_2); \qquad \textbf{(Z)}$$
$$e(\langle V\rangle_1, \langle W\rangle_2) \cdot e(\langle Y\rangle_1, \langle 1\rangle_2)^{-1} = e(\langle H\rangle_1, \langle r_y t\rangle_2). \qquad \textbf{(H)}$$

(where $\langle V\rangle_1 = \langle V_1\rangle_1 + \ldots + \langle V_n\rangle_1$, $\langle W\rangle_2 = \langle W_1\rangle_2 + \ldots + \langle W_n\rangle_2$, $\langle Y\rangle_1 = \langle Y_1\rangle_1 + \ldots$)

Fig. 1. Pinocchio-Based Adaptive zk-SNARK $(\mathsf{G}^1, \mathsf{P}^1, \mathsf{V}^1)$

QAP solution, which is enforced by providing $\langle z_j\rangle_1 = \langle\beta_i \cdot (v_j + w_j + y_j)\rangle_1$ elements only for the indices j of those variables. Our core idea is use external commitments of the form $\sum_k \boldsymbol{v}_k \cdot \langle x^k\rangle_1$ (that can be re-used across Pinocchio computations) and link the kth component of this commitment to the jth variable of the block using a modified $\langle z_j\rangle_1 = \langle\beta_i \cdot (x^k + v_j + w_j + y_j)\rangle_1$. We use one block per external commitment that the proof refers to. The witness (which is not committed to externally) is included in the first block, with the normal

Pinocchio element $\langle z_j \rangle_1 = \langle \beta_1 \cdot (v_j + w_j + y_j) \rangle_1$ just checking internal consistency as usual. The verification procedure changes slightly: $\langle V \rangle_1$ is no longer extended to $\langle V^+ \rangle_1$ to include public I/O (which we do not have); instead, the (**Z**) check ensures consistency with the corresponding commitment, for which there is a new correctness check (**C**).

The precise construction is shown in Fig. 1. This construction contains details on how to add randomness to make the proof zero-knowledge; and it shows how additional $\langle \alpha \cdot \rangle_1$ elements are added to obtain an extractable trapdoor commitment family $(\mathsf{G0}^1, \mathsf{Gc}^1, \mathsf{C}^1)$. In [14], we show that:

Theorem 1. *Under the* $(4d+3)$*-PDH, d-PKE, and* $(8d+6)$*-SDH assumptions (with d the maximal QAP degree),* $(\mathsf{G0}^1, \mathsf{Gc}^1, \mathsf{C}^1)$ *is an extractable trapdoor commitment scheme family, and* $(\mathsf{G}^1, \mathsf{P}^1, \mathsf{V}^1)$ *is an adaptive zk-SNARK.*

4 Smaller Proofs and Comparison to Literature

We now present two optimization that decrease the size of the above zk-SNARK, and compare the concrete efficiency of our three proposals to two related proposals from the literature. Note that, in the above construction, seven Pinocchio proof elements $\langle V \rangle_1$, $\langle \alpha_v V \rangle_2$, $\langle W \rangle_2$, $\langle \alpha_w W \rangle_1$, $\langle Y \rangle_1$, $\langle \alpha_y Y \rangle_2$, $\langle Z \rangle_1$ are repeated for each input commitment. We present two different (but, unfortunately, mutually incompatible) ways in which this can be avoided.

In our first optimization, inspired by a similar proposal to reduce verification work in Pinocchio ([3], later corrected by [12]), we decrease proof size and verification time at the expense of needing a larger-degree QAP. Namely, suppose that all variables in a given commitment occur only in the right-hand side of QAP equations. In this case, $v_j(x) = w_j(x) = 0$ for all j, so proof elements $\langle V_i \rangle_1, \langle \alpha_v V_i \rangle_2, \langle W_i \rangle_2, \langle \alpha_w W_i \rangle_1, \langle Y_i \rangle_1, \langle \alpha_y Y_i \rangle_2$ contain only randomness and, setting $\delta_{v,j} = \delta_{w,j} = 0$, can be omitted. As a consequence, the marginal costs per commitment used decrease from 7 to 3; the (**V**) and (**W**) verification steps can be skipped and the (**Z**) step simplified. To guarantee that a committed variable a only occurs in the right-hand of equations, we can introducing a witness b and equation $0 \cdot 0 = a - b$, slightly increasing the overall QAP size and degree. (This cannot be done for the first commitment since $\langle V_1 \rangle_1, \ldots$ also contain the witness, which occur in the left-hand side of equations as well.)

Our second proposal is a modified zk-SNARK that also reduces the marginal cost per commitment from 7 to 3, but gives more efficient verification when using many commitments. The core idea is to first concatenate all commitments $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$ into one "intermediate commitment $\boldsymbol{u}'$, and then use our original zk-SNARK with respect to $\boldsymbol{u}'$. More precisely, we build intermediate commitment $\boldsymbol{u}'_1$ with the first ℓ_1 values of $\boldsymbol{u}_1$; $\boldsymbol{u}'_2$ with ℓ_1 zeros followed by the first ℓ_2 values of $\boldsymbol{u}_2$; etcetera. Then, $\boldsymbol{u}' = \sum_i \boldsymbol{u}'_i$ is a commitment to the first $\ell_1, \ldots, \ell_n$ values of the respective commitments $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$. To avoid ambiguity between normal and intermediate commitments, to normal commitments we add a random factor r_c, i.e. $(r\langle r_c \rangle_1 + \sum_i \boldsymbol{v}_i \langle r_c x^i \rangle_1, r\langle \alpha r_c \rangle_2 + \ldots)$ and intermediate commitments are

as above[5]. Proving correspondence between normal and intermediate commitments is done similarly to the (**Z**) check above: we generate random β_i and give $\langle \beta'_i \cdot (r_c x^j + x^{\ell_1+\ldots+\ell_{i-1}+j} \rangle_1$ to the prover, who needs to produce proof element $\langle Z'_i \rangle_1$ such that $\langle Z'_i \rangle_1 = \beta'_i \cdot (\langle C_i \rangle_1 + \langle C'_i \rangle_1)$, which he can only do if $\langle C'_i \rangle_1$ is formed correctly. Details and the security proof appear in [14].

Table 1. Comparison between Pinocchio-based SNARKs (n: number of commitments; d is QAP degree; $d' \leq d$ is QAP degree with optimization; $D \geq d$ is fixed QAP degree)

Construction	Comm. size	Proof size	Prover computation		Verif comp
			Non-crypt. op	Crypt. op	
Geppetto	3 gr. el	8 gr. el	$\Theta(D \log D)$	$\Theta(D)$	$4n + 12$ pair.
Hash First+Pinocchio	2 gr. el	9n+1 gr. el	$\Theta(d \log d)$	$\Theta(d)$	$13n + 3$ pair.
Hash First+Pinocchio[a]	2 gr. el	5n+5 gr. el	$\Theta(d' \log d')$	$\Theta(d')$	$8n + 8$ pair.
Our zk-SNARK I	2 gr. el	7n+1 gr. el	$\Theta(d \log d)$	$\Theta(d)$	$11n + 3$ pair.
Our zk-SNARK I[a]	2 gr. el	3n+5 gr. el	$\Theta(d' \log d')$	$\Theta(d')$	$7n + 7$ pair.
Our zk-SNARK II	2 gr. el	3n+8 gr. el	$\Theta(d \log d)$	$\Theta(d)$	$6n + 12$ pair

[a] First optimization from Sect. 4 applied.

In Table 1, we provide a detailed comparison of our zk-SNARKs with two similar constructions: the Geppetto protocol due to [3] (which is also zero-knowledge but not adaptive); and the "hash first" approach applied to Pinocchio [7] (which is adaptive but not zero-knowledge). Geppetto is Protocol 2 from [3]. We assume QAP witnesses of $\mathcal{O}(d)$. In Geppetto, a fixed set of QAPs of degree d_i are combined into one large "MultiQAP" of degree D slightly larger than $\max d_i$. As a consequence, if both small and large computations need to be applied on the same data, then the small computations take over the much worse performance of the large computations. For Hash First+Pinocchio, we took the extractable scheme $\mathsf{XP}_{\mathcal{E}}$ since the Geppetto and our construction are extractable as well. To make it work on multiple commitments (which is described for neither Hash First nor Pinocchio), we assume natural generalisations of Hash First and of Pinocchio along the lines of [3,13]. Our first optimization can be applied to this construction; we mark the result with a star and write $d' \geq d$ for the increased degree due to the use of this optimization. Finally, we show our zk-SNARK without and with the first optimization; and our second zk-SNARK construction (to which the optimization does not apply).

In conclusion, Geppetto is the most efficient construction, but apart from not being adaptive, it also requires all computations to be fixed and of the same size, making it inefficient for small computations when they are combined with large ones. Our construction outperforms Hash First+Pinocchio, essentially adding zero knowledge for free; which variant is best depends on n and $d' - d$. Note that Hash First allows using the same commitment in different zk-SNARK schemes;

[5] Hence this construction can only handle inputs of combined size at most d.

our scheme only allows this for zk-SNARKs based on the kind of polynomial commitments used in Pinocchio.

5 Secure/Correct Adaptive Function Evaluation

In this section, we sketch how our zk-SNARK can be used to perform "adaptive function evaluation": privacy-preserving verifiable computation on committed data. We consider a setting in which multiple mutually distrusting *data owners* want to allow privacy-preserving outsourced computations on their joint data. A *client* asks a computation to be performed on this data by a set of *workers*. The input data is sensitive, so the workers should not learn what data they are computing on (assuming up to a maximum number of workers are passively corrupted). On the other hand, the client wants to be guaranteed the computation result is correct, for instance, with respect to a commitment to the data published by the data owner (making no assumption on which data owners and/or workers are actively corrupted). The difference in assumptions for the privacy and correctness guarantees is motivated by settings where data owners together choose the computation infrastructure (so they feel active corruption is unlikely) but need to convince an external client (e.g. a medical reviewer) of correctness. We work in the CRS model, where a trusted party (who is otherwise not involved in the system) performs one-time key generation.

In [14], we provide a precise security model that captures the above security guarantees by ideal functionalities. We define two ideal functionalities. The first guarantees privacy *and* correctness (*secure adaptive function evaluation*), and is realised by our construction if at most a threshold of workers are passively corrupted (but all other parties can be actively corrupted). The second guarantees only correctness (*correct adaptive function evaluation*), and is realised by our construction regardless of corruptions.

5.1 Our Construction

We now present our general construction based on multi-party computation and *any* adaptive zk-SNARK (as we will see later, our adaptive zk-SNARK gives a particularly efficient instantiation). At a high level, to achieve secure adaptive function evaluation, the workers compute the function using multi-party computation (MPC), guaranteeing privacy and correctness under certain conditions. However, when these conditions are not met, we still want to achieve correct adaptive function evaluation, i.e., we still want to ensure a correct computation result. To achieve this, the workers also produce, using MPC, a zk-SNARK proof of correctness of the result.

We require a MPC protocol in the outsourcing setting, i.e., with separate inputters (in our case, the data owners and the client), recipients (the client) and workers. The protocol needs to be reactive, so that the data owners can

provide their input before knowing the function to be computed[6]; and secure even if any number of data owners the client are actively corrupted. Security of the MPC protocol will generally depend on how many workers are corrupted; our construction will realise secure adaptive function evaluation (as opposed to just correct adaptive function evaluation) exactly when the underlying MPC protocol is secure. (As we show below, MPC protocols based on (t, n)-Shamir secret sharing (e.g., [5]) between $n = 2t + 1$ workers satisfy these requirements.)

Protocol Adaptive Trinocchio

(Data provider has $\boldsymbol{a_i} \in \mathbb{F}^d$; client has $\boldsymbol{a_c} \in \mathbb{F}^{d'}$, function $f : (\mathbb{F}^d)^n \times \mathbb{F}^{d'} \to \mathbb{F}^{d-d'}$.)

1. The trusted party generates a system-wide CRS crs of the trapdoor commitment family, and commitment keys $\mathsf{ck}_1, \ldots, \mathsf{ck}_n, \mathsf{ck}_c$ for the data owner and client. This material is distributed to all parties.
2. Each data owner computes commitment $c_i = \mathsf{C}_{\mathsf{ck}_i}(\boldsymbol{a_i}, r_i)$ to its input $\boldsymbol{a_i} \in \mathbb{F}^d$ using randomness r_i, and publishes it on a bulletin board.
3. The data owners, workers, and client use the MPC protocol to do the following:
   - Each data owner provides input $\boldsymbol{a_i}$ and randomness r_i
   - For each i, compute $c'_i = \mathsf{C}_{\mathsf{ck}_i}([\![\boldsymbol{a_i}]\!], [\![r_i]\!])$; if $c_i \neq c'_i$ then abort
4. The client provides function f to the trusted party. The trusted party determines a QAP Q computing f and a function f' solving Q, and performs key generation of the adaptive zk-SNARK (where one commitment combines the client's input and output). The client gets verification key crsv; the workers get Q, f', and the corresponding evaluation key crsp.
5. The data owners, workers, and client continue with the MPC from step 3:
   - Client: provide input $\boldsymbol{a_c}$
   - Compute $([\![\boldsymbol{b}]\!]; [\![\boldsymbol{w}]\!]) \leftarrow f'([\![\boldsymbol{a_1}]\!]; \ldots; [\![\boldsymbol{a_n}]\!]; [\![\boldsymbol{a_c}]\!])$
   - Compute $[\![c_c]\!] \leftarrow \mathsf{C}_{\mathsf{ck}_c}([\![\boldsymbol{a_c}]\!], [\![\boldsymbol{b}]\!]; [\![r_c]\!])$ for random r_c
   - Compute $[\![\pi]\!] \leftarrow \mathsf{P}(\mathsf{crs}, \{\mathsf{ck}_i\}, \ldots, \mathsf{ck}_n, \mathsf{ck}_c, \mathsf{crsp}; c_1, \ldots, c_n, [\![c_c]\!]; [\![\boldsymbol{a_1}]\!]; \ldots; [\![\boldsymbol{a_n}]\!]; [\![\boldsymbol{a_c}]\!], [\![\boldsymbol{b}]\!]; [\![r_1]\!], \ldots, [\![r_n]\!], [\![r_c]\!]; [\![\boldsymbol{w}]\!])$
   - Open outputs $[\![\boldsymbol{b}]\!], [\![r_c]\!], [\![c_c]\!], [\![\pi]\!]$ to the client
6. The client checks whether $\mathsf{V}(\mathsf{crs}, \mathsf{ck}_1, \ldots, \mathsf{ck}_n, \mathsf{ck}_c, \mathsf{crsv}; c_1, \ldots, c_n, c_c; \pi) = 1$ and $c_c = \mathsf{C}_{\mathsf{ck}_c}(\boldsymbol{a_c}, \boldsymbol{b}; r_c)$ and if so, returns computation result $\boldsymbol{b}$.

Fig. 2. The adaptive Trinocchio protocol

Our protocol is shown in Fig. 2. It uses an MPC protocol with the above properties, a trapdoor commitment family, and an adaptive zk-SNARK, instantiated for the function to be computed. The protocol relies on a trusted party that generates the key material of the zk-SNARK, but is otherwise not involved in the computation. Each data owner has an input $\boldsymbol{a_i} \in \mathbb{F}^d$ and the client has an

[6] Using non-reactive MPC requires is also possible, but then steps 3 and 4 of the protocol need to be swapped. As a consequence, data owners can abort based on the client's choice of function, leading to a weaker form of correct function evaluation.

input $\boldsymbol{a_c} \in \mathbb{F}^{d'}$ and a function $f : (\mathbb{F}^d)^n \times \mathbb{F}^{d'} \rightarrow \mathbb{F}^{d-d'}$ that it wants to compute on the combined data. Internal variables of the MPC protocol are denoted $[\![\cdot]\!]$.

In step 1, the trusted party sets up the trapdoor commitment family, generating separate keys for data providers and the client. (This prevents parties from copying each other's input.) In step 2, each data provider publishes a commitment to his input. In step 3, each data providers inputs its data and the randomness used for the commitment to the MPC protocol. The workers re-compute the commitments based on this opening and abort in case of a mismatch. (This prevents calling P on mismatching inputs in which case it may not be zero-knowledge.) In step 4, the client chooses the function f to be computed, based on which the trusted party performs key generation. (By doing this after the data owners' inputs, we prevent a selective failure attack from their side.) In step 5, the computation is performed. Using MPC, the client's output and witness are computed; a commitment to the client's I/O is produced, and a zk-SNARK proof of correctness with respect to the commitments of the data owners and client is built.[7] The client learns the output, randomness for its commitment, the commitment itself, and the proof. In step 6, the client re-computes the commitment and verifies the proof; in case of success, it accepts the output.

By sharing commitments between proofs, it is possible to generate key material for a number of small building blocks once, and later flexibly combine them into larger computations without requiring new key material. In particular, as we show in the case study, this enables computations on arbitrary-length data using the same key material (which was impossible in Trinocchio). It is also easy to support multiple clients or multiple commitments per data owner.

In [14], we show that indeed, the above construction achieves the formal definitions of secure adaptive function evaluation (under the same conditions of the corruptions of workers as the underlying MPC protocol) and correct adaptive function evaluation (regardless of corruptions).

5.2 Efficient Instantiation Using Secret Sharing and Our zk-SNARK

We now show that our zk-SNARKs and MPC based on Shamir secret sharing give a particularly efficient instantiation of the above framework. The idea is the same as for Trinocchio [13]: our zk-SNARK is essentially an arithmetic circuit of multiplicative depth 1, so given a solution to the QAP, the prover algorithm can be performed under MPC without any communication between the workers.

In more detail, we perform MPC based on Shamir secret sharing between the m workers (e.g., [5]). This guarantees privacy as long as at most t workers are passively corrupted, where $m = 2t + 1$. Inputs are provided by the inputters as an additive sharing between all workers: this way actively corrupted inputters cannot provide an inconsistent sharing. The workers Shamir-share and sum up the additive shares to obtain a Shamir sharing of the input. Outputs are provided

[7] Equivalently, the workers can open c_c and π and send them to the client in the plain.

to recipients either as Shamir shares or as freshly randomised additive shares: the latter allows producing our zk-SNARK proof without any communication.

Either of our zk-SNARK constructions can be used; we provide details for the first one. Below, write $[\![\cdot]\!]$ for Shamir sharing and $[\cdot]$ for additive sharing. (Note that Shamir sharings can be converted locally to additive sharings at no cost.) In step 3 of the protocol, to open c'_i, the parties apply $\mathsf{C}_{\mathsf{ck}_i}$ on their additive shares of the input and randomness, add a random additive sharing of zero (which can be generated non-interactively using pseudo-random zero sharing), and reveal the result. In step 5, $[\![\boldsymbol{b}]\!]; [\![\boldsymbol{w}]\!]$ are computed as Shamir secret shares. Next, $[c_c]$ is computed as an additive sharing by applying $\mathsf{C}_{\mathsf{ck}_c}$ on additive shares and adding a random sharing of zero. Next, P^1 is applied by performing the following steps:

- Generate $\delta_{v,i}, \delta_{w,i}, \delta_{y,i}$ by pseudo-random secret sharing.
- Compute $[\boldsymbol{h}] = ((\sum_j [\![\boldsymbol{x}_j]\!] \cdot v_j(x) + [\![\delta_v]\!] \cdot t(x)) \cdot (\sum_j [\![\boldsymbol{x}_j]\!] \cdot w_j(x) + [\![\delta_w]\!] \cdot t(x)) - (\sum_j [\![\boldsymbol{x}_j]\!] \cdot y_j(x) + [\![\delta_y]\!] \cdot t(x)))/t(x)$. Essentially this is done by performing the computation straight on Shamir secret shares; because there is only layer of multiplications of shares, this directly gives an additive sharing of the result. Smart use of FFTs gives time complexity $\mathcal{O}(d \cdot \log d)$ [2,13].
- All proof elements are now linear combinations of secret-shared data; compute them by taking linear combinations of the (Shamir or additive) shares and adding a random sharing of zero.

What remains is how to compute the solution of the QAP using multi-party computation. Namely, in addition to computing the function result $[\![\boldsymbol{b}]\!]$, the MPC also needs to compute witness $[\![\boldsymbol{w}]\!]$ to the QAP. Actually, if the function to be computed is described as an arithmetic circuit, this is very easy. Namely, in this case, the witness for the natural QAP for the function is exactly the vector of results of all intermediate multiplications; and these results are already available as Shamir secret shares as a by-product of performing the MPC. Hence, in this case, computing $[\![\boldsymbol{w}]\!]$ in addition to $[\![\boldsymbol{b}]\!]$ incurs no overhead.

If a custom MPC protocol for a particular subtask is used, then it is necessary to devise specific QAP equations and an MPC protocol to compute their witness. As an example, consider the MPC operation $[\![b]\!] \leftarrow [\![a \neq 0]\!]$, i.e., b is assigned 1 if $a \neq 0$ and 0 if $a = 0$. For computing $[\![b]\!]$, a fairly complex protocol is needed, cf. [5]. However, proving that b is correct using a QAP is simple [11]: introduce witnesses $c := (a + (1 - b))^{-1}$, $d := 1$ and equations:

$$a \cdot c = b \qquad a \cdot (d - b) = 0 \qquad d \cdot d = d.$$

Indeed, if $a = 0$ then the first equation implies that $b = 0$; if $a \neq 0$ then the second and third equations imply that $b = 1$. In both cases, the given value for c and $d = 1$ make all three equations hold. In our case study, we show similarly how, for complex MPC protocols for fixed-point arithmetic, simple QAPs proving correctness exist with easily computable witnesses.

6 Prototype and Distributed Medical Research Case

In this section, we present a proof-of-concept implementation of our second zk-SNARK construction and our Adaptive Trinocchio protocol. Computations can

be specified at a high level using a Python frontend; executed either locally or in a privacy-preserving way using multi-party computation; and then automatically proven and verified to be correct by a C++ backend. We show how two different computations can be performed on the same committed data coming from multiple data owners (with key material independent from input length, and optionally in a privacy-preserving way): aggregate survival statistics on two patient populations, and the "logrank test": a common statistical test whether there is a statistically significant difference survival rate between the populations.

6.1 Prototype of Our zk-SNARK and Adaptive Trinocchio

Our prototype, available at https://github.com/meilof/geppetri, is built on top of VIFF, a Python implementation of MPC based on Shamir secret sharing. In VIFF, computations on secret shares are specified as normal computations by means of operator overloading, e.g., assigning `c=a*b` induces a MPC multiplication protocol. We add a new runtime to VIFF that also allows computations to be performed locally without MPC.

To support computation proofs, we developed the `viffvc` library that provides a new data type: `VcShare`, a wrapper around a secret share. Each `VcShare` represents a linear combination of QAP variables. Addition and multiplication by constants of `VcShare`s is performed locally by manipulating the linear combination. Constants v are represented as $v \cdot one$, where witness one satisfies $one \cdot one = one$ so $one = 1$. When two `VcShare`s $\lambda_1 x_1 + \ldots$ and $\mu_1 x_1 + \ldots$ are multiplied, a local or MPC multiplication operation is performed on the underlying data, and the result is a new `VcShare` x_k wrapping the result as a new QAP variable. QAP equation $(\lambda_1 x_1 + \ldots) \cdot (\mu_1 x_1 + \ldots) = 1 \cdot x_k$ is written to a file, and the multiplication result x_k or its secret share, when known, is written to another file. Apart from multiplication, some additional operations are supported. For the $[\![b]\!] \leftarrow [\![a \neq 0]\!]$ operation discussed in Sect. 5.2, the implementation computes $[\![b]\!]$ and $[\![c]\!] = ([\![a]\!] + (1 - [\![b]\!]))^{-1}$, and writes these secret shares/values and the equations from Sect. 5.2 to the respective files. We also support secret indexing (e.g., [5]), and fixed-point computations as discussed below.

Computations are performed by this custom VIFF-based system together with an implementation of our zk-SNARK. A first tool, `qapgen`, generates the CRS for our trapdoor commitment scheme. A second tool, `qapinput`, builds a commitment to a given input; and computes secret shares of these inputs that are used for MPC computations. Then, our Python implementation is used to compute the function, either locally or using multi-party computation. At the end of this execution, there is one file with the QAP equations, and one file with values/shares for each QAP variable. Our `qapgenf` tool uses the first file to perform key generation of the QAP (this is done only once and for next executions, previous key material is re-used). Our `qapprove` tool uses the second file to generate the zk-SNARK proof (shares) to be received by the client. Finally, a `qapver` tool verifies the proof based on the committed inputs and outputs.

Algorithm 1. Anonymized survival data computation

Require: $[\![\boldsymbol{d_1}]\!], [\![\boldsymbol{n_1}]\!], [\![\boldsymbol{d_2}]\!], [\![\boldsymbol{n_2}]\!]$: block of survival data points for two populations
Ensure: $([\![d'_1]\!], [\![n'_1]\!], [\![d'_2]\!], [\![n'_2]\!])$ aggregated survival data for the block
1: **function** SUMM$([\![\mathbf{d}_{i,1}]\!], [\![\mathbf{d}_{i,2}]\!], [\![\mathbf{n}_{i,1}]\!], [\![\mathbf{n}_{i,2}]\!])$
2: **return** $(\sum_i [\![\boldsymbol{d_{1,i}}]\!], [\![\boldsymbol{n_{1,1}}]\!], \sum_i [\![\boldsymbol{d_{2,i}}]\!], [\![\boldsymbol{n_{2,1}}]\!])$

6.2 Application to Medical Survival Analysis

We have applied our prototype to (adaptively) perform computations on survival data about two patient populations. In medical research, survival data about a population is a set of tuples (n_j, d_j), where n_j is the number of patients still in the study just before time j and d_j is the number of deaths at time j. We assume both populations are distributed among multiple hospitals, that each commit to their contributions $(d_{j,1}, n_{j,1}, d_{j,2}, n_{j,2})$ to the two populations at each time.

Aggregate Survival Data. Our first computation is to compute an aggregate version of the survival data, where each block $\{d_{j,1}, n_{j,1}, d_{j,2}, n_{j,2}\}_{j=1}^{25}$ of 25 time points is summarised as $(\sum_j d_{j,1}, n_{1,1}, \sum_j d_{j,2}, n_{1,2})$. The function SUMM computing this summary is shown in Algorithm 1. Function SUMM translates into a QAP on 26 commitments: as input, for each time point j, a commitment $\sum_i c_{i,j}$ to the combined survival data $([\![\boldsymbol{d_{i,1}}]\!], [\![\boldsymbol{n_{i,1}}]\!], [\![\boldsymbol{d_{i,2}}]\!], [\![\boldsymbol{n_{i,2}}]\!])$ from the different hospitals i at that time (using the fact that commitments are homomorphic); as output, a commitment to $([\![d'_1]\!], [\![n'_1]\!], [\![d'_2]\!], [\![n'_2]\!])$.

Logrank Test. Our second computation is the so-called "Mantel-Haenzel logrank test", a statistical test to decide whether there is a significant difference in survival rate between the two populations (as implemented, e.g., in R's `survdiff` function). Given the survival data from two populations, define:

$$E_{j,1} = \frac{(d_{j,1} + d_{j,2}) \cdot n_{j,1}}{n_{j,1} + n_{j,2}}; \quad V_j = \frac{n_{j,1} n_{j,2} (d_{j,1} + d_{j,2})(n_{j,1} + n_{j,2} - d_{j,1} - d_{j,2})}{(n_{j,1} + n_{j,2})^2 \cdot (n_{j,1} + n_{j,2} - 1)};$$

$$X = \frac{\sum_j E_{j,1} - \sum_j d_{j,1}}{\sum_j V_j}.$$

The null hypothesis for the logrank test, i.e., the hypothesis that the two curves represent the same underlying "survival function", corresponds to $X \sim \chi^2_1$. This null hypothesis is rejected (i.e., the curves are different) if $1 - \mathrm{cdf}(X) > \alpha$, where cdf is the cumulative density function of the χ^2_1 distribution and, e.g., $\alpha = 0.05$. We use MPC to compute X, and then apply the cdf in the clear.

Our implementation consists of two different functions: a function BLOCK (Algorithm 2) that computes $(E_{j,1}, V_j, d_{j,1})$ given the survival data at point j; and a function FIN that, given $\sum E_{j,1}$, $\sum V_j$, and $\sum d_{j,1}$ computes X (Algorithm 3). As above, function BLOCK is applied to commitment $\sum_i c_{i,j}$ to the combined survival data from different hospitals at a particular time, giving output commitment c'_j. Function FIN is applied to commitment $\sum_j c'_j$ to

Algorithm 2. Logrank computation for each time step

Require: $[\![\mathbf{d}_{i,1}]\!], [\![\mathbf{d}_{i,2}]\!], [\![\mathbf{n}_{i,1}]\!], [\![\mathbf{n}_{i,2}]\!]$ survival data at time point i
Ensure: $([\![e_i]\!]^f, [\![v_i]\!]^f, [\![d_i]\!])$ contributions to $\sum_j E_{j,1}$, $\sum_j V_j$, $\sum_j d_{j,1}$ for test statistic

function BLOCK($[\![\mathbf{d}_{i,1}]\!], [\![\mathbf{d}_{i,2}]\!], [\![\mathbf{n}_{i,1}]\!], [\![\mathbf{n}_{i,2}]\!]$)
  $[\![ac]\!] \leftarrow [\![\mathbf{d}_{i,1}]\!] + [\![\mathbf{d}_{i,2}]\!]$
  $[\![bd]\!] \leftarrow [\![\mathbf{n}_{i,1}]\!] + [\![\mathbf{n}_{i,2}]\!]$
  $[\![frc]\!]^f \leftarrow [\![ac]\!]/[\![bd]\!]$
  $[\![e_i]\!]^f \leftarrow [\![frc]\!]^f \cdot [\![\mathbf{n}_{i,1}]\!]$
  $[\![vn]\!] \leftarrow [\![\mathbf{n}_{i,1}]\!] \cdot [\![\mathbf{n}_{i,2}]\!] \cdot [\![ac]\!] \cdot ([\![bd]\!] - [\![ac]\!])$
  $[\![vd]\!] \leftarrow [\![bd]\!] \cdot [\![bd]\!] \cdot ([\![bd]\!] - 1)$
  $[\![v_i]\!]^f \leftarrow [\![vn]\!]/[\![vd]\!]$
  return $([\![e_i]\!]^f, [\![v_i]\!]^f, [\![d_i]\!])$

Algorithm 3. Logrank final computation

Require: $[\![es]\!], [\![vs]\!], [\![ds]\!]$: summed-up values required to compute X
Ensure: $[\![chi]\!]^f$ test statistic comparing two curves; supposedly $chi \sim \chi^2_1$

function FIN($[\![es]\!], [\![vs]\!], [\![ds]\!]$)
  $[\![ds]\!]^f \leftarrow [\![ds]\!] \ll \text{PRECISION}$
  $[\![dmi]\!]^f \leftarrow [\![ds]\!]^f - [\![vs]\!]^f$
  $[\![chi0]\!]^f \leftarrow [\![dmi]\!]^f / [\![vs]\!]^f$
  $[\![chi]\!]^f \leftarrow [\![chi0]\!]^f \cdot [\![dmi]\!]^f$
  return $[\![chi]\!]^f$

$(\sum E_{j,1}, \sum V_j, \sum d_{j,1})$, again using the fact that commitments are homomorphic; outputting a commitment to X that is output to the client.

Algorithms 2 and 3 use fixed-point numbers $[\![x]\!]^f$, representing value $x \cdot 2^{-k}$ where we use precision $k = 20$. We use the fixed-point multiplication $[\![c]\!]^f \leftarrow [\![a]\!]^f \cdot [\![b]\!]^f$ and division $[\![c]\!]^f \leftarrow [\![a]\!]/[\![b]\!]$, $[\![c]\!]^f \leftarrow [\![a]\!]^f/[\![b]\!]^f$ protocols due to [5]. To prove that $[\![c]\!]^f \leftarrow [\![a]\!]^f \cdot [\![b]\!]^f$ is correct, note that we need to show that $2^k c - a \cdot b \in [-2^k, 2^k]$, or equivalently, that $\alpha := 2^k c - a \cdot b + 2^k \geq 0$ and $\beta := 2^k - (2^k c - a \cdot b) \geq 0$. We prove this by computing, using MPC, bit decompositions [5] $\alpha = \alpha_0 + \alpha_1 \cdot 2 + \ldots + \alpha_k \cdot 2^k$ and $\beta = \beta_0 + \beta_1 \cdot 2 + \ldots + \beta_k \cdot 2^k$ (indeed, α and β are $\leq k+1$ bits long); these α_i, β_i are the witnesses to QAP equations:

$$\forall i : \alpha_i \cdot (1 - \alpha_i) = 0 \qquad c - a \cdot b + 2^k = \alpha_0 + \alpha_1 \cdot 2 + \ldots + \alpha_k \cdot 2^k$$
$$\forall i : \beta_i \cdot (1 - \beta_i) = 0 \qquad \beta = 2^k - (c - a \cdot b) = \beta_0 + \beta_1 \cdot 2 + \ldots + \beta_k \cdot 2^k.$$

Similarly, note that $[\![c]\!]^f \leftarrow [\![a]\!]^f/[\![b]\!]^f$ is correct if and only if $2^k a - b \cdot c \in [-b, b]$, i.e., $\gamma := b + 2^k a - b \cdot c \geq 0$ and $\delta := b - (2^k a - b \cdot c) \geq 0$. If b has bitlength at most K (i.e., the represented number has absolute value $\leq 2^{K-k}$), then γ and δ have at most $K+1$ bits. As above, we prove correctness by determining $(K+1)$-length bit decompositions of γ and δ and proving them correct. Proving correctness of $[\![c]\!]^f \leftarrow [\![a]\!]/[\![b]\!]$ is analogous.

Performance. Table 2 shows the performance of our proof-of-concept implementation for computing aggregate survival data and the logrank test (on a modern

Table 2. Performance: computation/proving/verification; with/without MPC

Aggregate	Computation (function): 0.0 s (w/o MPC)/0.1 s (w/MPC) Computation (function+witness): 0.0 s (w/o MPC)/0.1 s (w/MPC)		
	BS=1	BS=25	BS=175
	QAP degree: 3	QAP degree: 3	QAP degree: 57
	Prover: 0.3 s/0.4 s	Prover: 0.1 s/0.1 s	Prover: 0.0 s/0.0 s
	Verifier: 1.2 s/1.5 s	Verifier: 0.2 s/0.2 s	Verifier: 0.0 s/0.0 s
Logrank	Computation (function): 0.2 s (w/o MPC) / 190.5 s (w/MPC) Computation (function+witness): 0.6 s (w/o MPC)/235.2 s (w/MPC)		
	BS=1	BS=25	BS=175
	QAP deg (block): 173	QAP deg (block): 4304	QAP deg (block): 30104
	QAP deg (fin): 85	QAP deg (fin): 85	QAP deg (fin): 85
	Prover: 13.9 s/78.5 s	Prover: 16.2 s/81.0 s	Prover: 9.8 s/73.5 s
	Verifier: 3.9 s/4.9 s	Verifier: 0.2 s/0.3 s	Verifier: 0.0 s/0.0 s

laptop). As input, we used the "btrial" data set included in R's "kmsurv" package (on which we indeed reproduce R's `survdiff` result) of 175 data points. Apart from having one data point per commitment, we also experiment with having a "block size" of 25 or 175 data points. For the logrank test, we use one QAP per block; larger blocks mean less work for the verifier (since there are fewer proofs) but, in theory, more work for the prover (since the proving algorithm is superlinear in the QAP size). For aggregation, we use one QAP per 25 data points or per commitment, whichever is more.

We time the performance of running the computation, producing the proof, and verifying it, with or without MPC. As expected, MPC induces a large overhead for the computation, especially for the logrank test (due to the many fixed-point computations). MPC also incurs an overhead for proving: this is because of the many exponentiations with $|\mathbb{F}|$-sized secret shares rather than small witnesses. Note that proving is faster than computing with MPC: the underlying operations are slower [13], but the QAP proof is in effect on a verification circuit that is smaller than the circuit of the computation itself [6]. Proving is faster for block size 175 than block size 25, which is unexpected; this may be because our FFT subroutine rounds up QAP degrees to the nearest power of two, which is favourable in the 175-sized case but not in the 25-sized case. As expected, verification is faster for larger block sizes. (The overhead of MPC here is due to recombing the proof shares into one overall proof.)

7 Conclusion

In this work, we have given the first practical Pinocchio-based adaptive zk-SNARK; applied it in the privacy-presering outsourcing setting; and presented a proof-of-concept implementation. We mention a few promising directions for follow-ups. Concerning our construction for making Pinocchio adaptive, it would

be interesting to see if it can be applied to make recent, even more efficient zk-SNARKS (e.g., [9] in the generic group model) adaptive as well. Moreover, apart from providing a non-adaptive zk-SNARK, Geppetto also introduces the interesting idea of proof bootstrapping, where the verification procedure of the zk-SNARK itself can be performed by means of a verifiable computation, so multiple related proofs can be verified in constant time. Applying this technique in our setting should combine our flexibility with their constant-time verification.

Concerning our privacy-preserving outsourcing framework, it is interesting to see if, apart from secret sharing plus our SNARK, other appealing instantiations are possible. Also, the combination of MPC and verifiable computation raises the challenge to construct efficient QAPs for specific operations and build efficient MPC protocols for computing their witnesses. We have presented zero testing and fixed-point computations as examples, but the same idea is applicable to many other operations as well. More generally, extending our zk-SNARK prototype with more basic operations, and improving its user-friendliness, would help bring the techniques closer to practice.

Acknowledgements. This work is part of projects that have received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 643964 (SUPERCLOUD) and No. 731583 (SODA).

References

1. Backes, M., Barbosa, M., Fiore, D., Reischuk, R.M.: ADSNARK: nearly practical and privacy-preserving proofs on authenticated data. In: Proceedings S&P (2015)
2. 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. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40084-1_6
3. Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: versatile verifiable computation. In: Proceedings S&P, pp. 253–270 (2015)
4. Danezis, G., Fournet, C., Groth, J., Kohlweiss, M.: Square span programs with applications to succinct NIZK arguments. In: Sarkar, P., Iwata, T. (eds.) ASIACRYPT 2014. LNCS, vol. 8873, pp. 532–550. Springer, Heidelberg (2014). doi:10.1007/978-3-662-45611-8_28
5. de Hoogh, S.: Design of large scale applications of secure multiparty computation: secure linear programming. Ph.D. thesis, Eindhoven University of Technology (2012)
6. de Hoogh, S., Schoenmakers, B., Veeningen, M.: Certificate validation in secure computation and its use in verifiable linear programming. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 265–284. Springer, Cham (2016). doi:10.1007/978-3-319-31517-1_14
7. Fiore, D., Fournet, C., Ghosh, E., Kohlweiss, M., Ohrimenko, O., Parno, B.: Hash first, argue later: adaptive verifiable computations on outsourced data. In: Proceedings CCS (2016)

8. 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). doi:10.1007/978-3-642-38348-9_37
9. Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49896-5_11
10. Lipmaa, H.: Prover-efficient commit-and-prove zero-knowledge SNARKs. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 185–206. Springer, Cham (2016). doi:10.1007/978-3-319-31517-1_10
11. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: nearly practical verifiable computation. In: Proceedings S&P, pp. 238–252 (2013)
12. Parno, B.: A note on the unsoundness of vntinyram's snark. Cryptology ePrint Archive, Report 2015/437 (2015). http://eprint.iacr.org/
13. Schoenmakers, B., Veeningen, M., Vreede, N.: Trinocchio: privacy-preserving outsourcing by distributed verifiable computation. In: Manulis, M., Sadeghi, A.-R., Schneider, S. (eds.) ACNS 2016. LNCS, vol. 9696, pp. 346–366. Springer, Cham (2016). doi:10.1007/978-3-319-39555-5_19
14. Veeningen, M.: Pinocchio-based adaptive zk-SNARKS and secure/correct adaptive function evaluation. Cryptology ePrint Archive, Report 2017/013 (2017). http://eprint.iacr.org/2017/013

http://www.springer.com/978-3-319-57338-0

Progress in Cryptology - AFRICACRYPT 2017
9th International Conference on Cryptology in Africa,
Dakar, Senegal, May 24-26, 2017, Proceedings
Joye, M.; Nitaj, A. (Eds.)
2017, X, 231 p. 42 illus., Softcover
ISBN: 978-3-319-57338-0