SNARKs for Stateful Computations on Authenticated Data

Johannes Reinhart[1] Erik-Oliver Blass[2] Bjoern Annighoefer[1]

[1] University of Stuttgart, Stuttgart, Germany
{johannes.reinhart, bjoern.annighoefer}@ils.uni-stuttgart.de
[2] Airbus, Munich, Germany
erik-oliver.blass@airbus.com

Abstract. We present a new generalization of (zk-)SNARKs specifically designed for the application domain of safety-critical control systems. These need to be protected against adversarial tampering as well as non-malicious but unintended system failures due to random faults in components. Our SNARKs combine two additional features at the same time. Besides the verification of correct computation, they also allow, first, the verification of input data authenticity. Specifically, a verifier can confirm that the input to the computation originated from a trusted source. Second, our SNARKs support verification of stateful computations across multiple rounds, ensuring that the output of the current round correctly depends on the internal state of the previous round. Our focus is on concrete practicality, so we abstain from arithmetizing hash functions or signatures in our SNARKs. Rather, we modify the internals of an existing SNARK to extend its functionality. We implement and benchmark our new SNARKs in a sample scenario of a real-time high-integrity flight control system.
With our construction, prover runtime improves significantly over the baseline by a factor of 90. Verification time increases by 36 %, but is less than comparable approaches that do not arithmetize hash functions or signatures.

1 Introduction

In many applications, one party must demonstrate to others that they have correctly executed a certain computation. Specifically, party $\mathcal{P}$ takes input x and computes output $y = f(x)$ for some function f. To then prove correct computation of y, $\mathcal{P}$ (the prover) generates a proof π. The idea is that any verifier $\mathcal{V}$ can use π and check whether y is the correct output with respect to x and f. Two key properties towards practical efficiency are that verification of π should be more efficient than re-computing y, and that the size of π is small. The construction of such proofs is a very active area of research as indicated by the recent flurry of papers on, for example, succinct non-interactive ARguments of Knowledge (SNARKs) [1, 8, 10–12, 16, 22, 34, 36, 38, 40, 47, 52, 55, 56]. An additional attribute as provided by zero-knowledge SNARKs (zk-SNARKs) is that $\mathcal{V}$ does not learn anything about x besides the validity of the proof.

This paper explores a new generalization of (zk-)SNARKs for applications in digital control systems. A typical digital control system comprises a central control unit which computes control outputs for actuators to manipulate a technical system. In most such cases, the control unit must periodically process sensor input x_t, update its internal state s_t and provide output commands y_t. Many control systems are safety-critical, such that a malfunction or adversarial tampering can lead to loss of lives. Furthermore, modern control system have a large complexity and inter-connectivity, exposing many attack surfaces. Applying ADSC-SNARKs in such a setting can be used to prevent cyber-attacks on control units or communication channels or help detect incorrect outputs from malfunctioning devices. These settings would in particular benefit from the ability to verify output commands of a control unit immediately in each iteration t. Our SNARKs combine two key properties to support this use case.

SNARKs over Authenticated Data If $\mathcal{V}$ must not learn x for performance or privacy reasons, the natural question is how to authenticate x, such that a malicious $\mathcal{P}$ does not choose arbitrary x as input. So, the first additional property we consider in this paper is that x be authenticated by a third party, e.g., the sensor actually measuring physical quantity x_t. Consequently, $\mathcal{P}$ must additionally prove to $\mathcal{V}$ that input x to f has been authenticated by the third party before. A crucial challenge in this context is that $\mathcal{V}$ does not have access to the third party at the time of verification.

State-Consistent SNARKs The computation of a digital controller usually depends on a changing state. That is, given initial state s_1, a function f is applied iteratively on state s_t and input x_t producing output y_t and updated state s_{t+1}. So, in each iteration t,

$$(y_t, s_{t+1}) = f(x_t, s_t).$$

Again, the technical challenge comes in the situation where party $\mathcal{P}$ computing f is untrusted or unreliable. An adversarial $\mathcal{P}$ might arbitrarily change states s_t between iterations. Thus, the second property we target in this paper is for $\mathcal{P}$ to create a proof π_t allowing $\mathcal{V}$ to efficiently verify that y_t is the correct output when evaluating f on the state update of the previous iteration t. As the state is often large, and communication bandwidth or memory of the verifying party can be limited, we require that the size of π_t should also be smaller than the size of state s_t. Along the same lines of zero-knowledge regarding inputs, we also require a zero-knowledge property for the state. That is, $\mathcal{V}$ should not learn any information about a state beyond what can be inferred by the output y_t of the computation. In conclusion, in each iteration t, $\mathcal{P}$ proves to $\mathcal{V}$ the correctness of computing f, the use of authenticated input x_t, and the consistency of state s_t at the same time.

Related Work While we describe related work in detail later in Section 6, we briefly summarize the differences of our setting.

The state-consistency property is given in the setting of incrementally verifiable computation (IVC), which can be achieved with recursive SNARKs [9]

Table 1. Comparison of our ADSC-SNARK with related approaches. $|X|$: number of private inputs, $|S|$: number of states, $|\Omega|$: size of witness, $\mathbb{F}_p$: prime field element, $\mathbb{G}_1$: curve group 1 element, $\mathbb{G}_2$: curve group 2 element, *sig*: signature, $\mathrm{MSM}_{\mathbb{G}_1}(n)$: multi-scalar multiplication of n $\mathbb{G}_1$ elements, P: pairing evaluation, SV: signature verification, PV: public verifiability, AD: proof about authenticated data, SC: state consistency. Overhead means difference in runtime or size of new SNARK compared to its base SNARK. Both arithmetization and prover overhead increase prover runtime.

Approach	Base SNARK	Arith. Overhead	Prover Overhead	Verifier Overhead	Proof Overhead	PV	AD	SC
Folklore ("Strawman")	Groth16 [40]	2126 $72\|X\|$ $144\|S\|$	—	—	$1\times\mathbb{F}_p$	✓	✓	✓
AD-SNARK [3] (dv)	BCTV14 [10]	$\|X\|$	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|)$	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|)$ $2\times\mathrm{P}$	$3\times\mathbb{G}_1$	—	✓	—
AD-SNARK [3] (pv)	BCTV14 [10]	$\|X\|$	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|)$	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|)$ $(\|X\|+4)\times\mathrm{P}$ $\|X\|\times\mathrm{SV}$	$3\times\mathbb{G}_1$ $\|X\|\times\mathbb{G}_2$ $\|X\|\times sig$	✓	✓	—
SPHinx [33]	Marlin [22]	—	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|\Omega\|)$ $4\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|)$ $1\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|\|\Omega\|)$	$7\times\mathrm{P}$ $2\times\mathrm{MSM}_{\mathbb{G}_1}(\|X\|)$	$15\times\mathbb{G}_1$ $10\times\mathbb{F}_p$	✓	✓	—
Geppetto [24] SC-SNARK	Pinocchio [52]	$2\times\|S\|$	$3\times\mathrm{MSM}_{\mathbb{G}_1}(\|S\|)$	$1\times\mathrm{P}$	$3\times\mathbb{G}_1$	✓	—	✓
LegoGro [17] SC-SNARK	Groth16 [40]	$2\times\|S\|$	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|S\|)$ $+1\times\mathrm{MSM}_{\mathbb{G}_1}(2\|S\|)$	$4\times\mathrm{P}$	$3\times\mathbb{G}_1$	✓	—	✓
ADSC-SNARK **(ours)**	Groth16 [40]	—	$1\times\mathrm{MSM}_{\mathbb{G}_1}(\|S\|)$	$3\times\mathrm{P}$ $1\times\mathrm{SV}$	$2\times\mathbb{G}_1$ $1\times sig$	✓	✓	✓

or folding schemes [27, 43, 44]. These allow a verifier to check an entire iterative computation up to the current step with a single proof. However, this is a stronger requirement compared to our setting, where we only check a single iteration at once. IVC is achieved by proving in each iteration the correctness of a proof verification of the previous iteration or the correctness of compressing (folding) two computational steps.

Constructions for composing SNARKs [17, 24] use commit-and-prove SNARKs to prove the conjunction, disjunction or functional composition of two or more computations $f_1, f_2, ...$ These fit our setting better, because they can be used to prove stateful computations without an additional recursion or folding overhead. Our construction is therefore based on and compared to this approach. The composition still incurs overhead compared to a plain SNARK, this is shown in Table 1. The overhead includes increasing the size of the proven relation (arithmetization overhead), additional computations by the prover and the verifier or more proof elements.

Similarly, previous SNARKs for computations on authenticated data [3, 25, 33] are derived from a plain SNARK and come with some additional overhead as shown in Table 1. Unfortunately, there does not exist any construction that considers both stateful computations and computations on authenticated data. Combining both properties can be non-trivial. For example the designated-

verifier SNARK in [3] uses one-time message authentication codes for the input data, which would be insecure in an iterative setting without additional measures. Theoretically, it would be possible to prove an iterative computation by proving f for all steps t and all inputs x_t at once instead of in each iteration. However, this would be inefficient, as prover time would grow with each iteration. Furthermore, our specific setting allows for some performance optimizations, such that the overhead of our ADSC-SNARK is less than the combined overhead of two of the existing approaches.

Several works present proofs about authenticated data by arithmetizing and proving a signature verification algorithm [42, 54, 60]. Similarly, several works present proofs about computations with state by arithmetizing and proving hash functions [7, 15, 49, 54]. In practice, these approaches lead to long proving times due to arithmetization overhead. In this paper, we call approaches that arithmetize hashes or verification algorithms as *folklore* approaches.

This paper We present a new scheme dubbed ADSC-SNARK which allows to efficiently verify the correctness of each value y_t of a stateful computation on authenticated data, given a succinct proof in each iteration t.

Besides our conceptual contributions, we also implement ADSC-SNARK and compare it against related work, resulting in a concrete speedup of 89× over the folklore approach for a circuit with 2^{14} states and inputs. The C++ implementation is open source and freely available[1].

The **technical highlights** of this paper are:

- A formal definition of the security properties for ADSC-SNARKs, including definitions for completeness, soundness, and zero-knowledge.
- The first construction of an efficient, public verifier ADSC-SNARK that is significantly more performant in terms of prover runtime than folklore approaches where either a signature verification algorithm or a hashing algorithm is arithmetized.
  Crucially, proof size and verifier runtime are constant in the size of witness, input, and internal state.
- An open source C++ implementation of our ADSC-SNARK based on the standard libsnark library. This is particularly suitable for practitioners in domains where C++ is predominantly used such as in embedded systems engineering.
- An evaluation that benchmarks ADSC-SNARK and compares its performance to LegoGro16 by Campanelli et al. [17], AD − SNARK by Backes et al. [3] and the folklore approach with highly optimized signature verification and hashing arithmetizations. Our evaluation demonstrates that ADSC-SNARKs are not only asymptotically but also concretely practical.
- A proof-of-concept application and its evaluation in the form of a safety-critical control system, where actuator commands provided by an untrusted controller can be verified. This work is an important step towards a practical application of SNARKs in real-time settings.

[1] https://github.com/johannes-reinhart/adsc-snark

Zero-Knowledge As with related work [10, 40, 52], also our ADSC-SNARK construction proves in zero-knowledge by adding randomization masks to the proof. Due to space constraints and to ease exposition, we will focus on describing ADSC-SNARKs *without* the zero-knowledge property.Appendix B presents the full construction of ADSC-SNARKs *with* zero-knowledge. We stress that our implementation and evaluation in Section 4 is performed with the full version of ADSC-SNARK, including the zero-knowledge property.

1.1 Our solution in a nutshell

State Consistency: To prove that the state between two computations is consistent, $\mathcal{P}$ might send constant-sized hashes of the state $H(s_t)$ and the updated state $H(s'_t)$ to $\mathcal{V}$ and augment proof π_t to show consistency of $H(s_t)$ with $H(s'_{t-1})$ from the previous iteration. This turns out to be expensive, as the computation to be proven needs to be represented as a SNARK circuit. Additionally proving consistency of $H(s_t)$ with $H(s'_{t-1})$ requires arithmetization of cryptographic hash function H. Adding an arithmetized hash function increases the size of the SNARK circuit and therefore prover time significantly [7, 15, 49, 54].

To overcome this drawback, we follow the observation by Campanelli et al. [17] that the SNARK by Groth [40] can be strengthened to a *commit-carrying* SNARK. Essentially, the proof does not only prove the validity of a computation, but also commits to some parts of the data in the computation. This commitment can then be used as a link to another proof of a commit-carrying SNARK. Two such proofs can finally be checked whether they commit to the same data. While Campanelli et al. add further transformations to this commit-carrying SNARK to achieve compatibility between different proving schemes, we are able to directly link the commitments of two proofs of the same SNARK. This results in a significant reduction of proof size, verifier runtime, and prover runtime.

Authenticated Data: To prove that the computation has been carried out on authenticated input data x, one might add a signature to x and prove correctness of the signature with respect to x. Again, this would imply an expensive arithmetization of the signature verification algorithm.

Our key idea to overcome this issue is to only sign a commitment to the authenticated data. This commmitment can be linked to a commit-carrying SNARK, similarly to our techniques for state-consistency described above. In addition to verifying the proof π, $\mathcal{V}$ only needs to verify the signature on the commitment which can be independent of the size of the actual committed data. A similar idea was recently applied by Datta et al. [25] to make proofs on signed images. However, they base their construction on the Plonk [34] proof system, which results in a larger proof size compared to our construction.

1.2 Notation

In this paper, λ denotes the computational security parameter. For instantiating our scheme, we use prime-order groups $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, $|\mathbb{G}_1|$, $|\mathbb{G}_2|$, $|\mathbb{G}_T| \in poly(\lambda)$

together with a Type-3 bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We denote a group element α in $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ by its discrete logarithm $[\alpha]_1$, $[\alpha]_2$, $[\alpha]_T$. The group operation in $\mathbb{G}_1$ ($\mathbb{G}_2, \mathbb{G}_T$ accordingly) is represented as addition, and scalar multiplication of element α by scalar b by $b\,[\alpha]_1 = [b\alpha]_1$. The multiplication operator $(\cdot)$ applied to group elements from $\mathbb{G}_1$ and $\mathbb{G}_2$ represents applying the bilinear map e to the operands. $\mathbb{F}^*$ denotes the set of invertible elements of a field $\mathbb{F}$.

For a randomized algorithm $F(x; r_\$)$ with input x and random coins $r_\$$, $y \leftarrow F(x)$ denotes random sampling of $r_\$$ and then assigning the output of $F(x; r_\$)$ to y. We denote the assignment of multiple elements using independent random coins $r_\$$ for F and set S as $y_t \leftarrow F(x_t)$ for $t \in S$. For two randomized algorithms F and H, we write $(y; z) \leftarrow (F||H)(x)$ for picking random coin $r_\$$ and then assigning the result of applying F to x with $r_\$$ to y, and assigning the result of applying H to the same input x and the same random coin $r_\$$ to z. We write $F^{G(\cdot)}$ to denote that F has oracle access to G. To sample an element from a set S uniformly from random, we write $y \leftarrow\!\!\$\, S$.

For functions f and g, the approximate equal operator $\approx$ denotes that the difference $f - g$ is negligible in security parameter λ:

$$f \approx g \Leftrightarrow \forall c \in \mathbb{N} : \exists\, \lambda_0 \text{ such that } \forall \lambda > \lambda_0 : |f(\lambda) - g(\lambda)| < \lambda^{-c}.$$

For a set S, we write $\mathrm{pos}_S(i)$ to order elements i by size, i.e., for $S = \{3, 4\}$, $\mathrm{pos}_S(3) = 1$, $\mathrm{pos}_S(4) = 2$. We write PPT for a probabilistic polynomial time algorithm and Pr for probability.

2 ADSC-SNARK Background

Before our main construction, we present formal definitions of an ADSC-SNARK and its properties for a family of relations R_λ and security parameter λ.

2.1 Setup

We assume that R_λ can be generated by a generator $\mathrm{pp} \leftarrow \mathsf{Gen}(\lambda)$ outputting public-parameters pp, such that pp include a field $\mathbb{H}$ and a relation $\mathrm{R} \in \mathrm{R}_\lambda$ over m variables a_i from field $\mathbb{H}$.

We require that relation R allows mapping each variable a_i into one of five different categories. These are represented by disjoint vectors ϕ, x, s, s', ω with corresponding index sets Φ, X, S, S', Ω as follows:

1. *public input-output* variables $\phi = (a_i)_{i \in \Phi}$,
2. *private input* variables $x = (a_i)_{i \in X}$,
3. *state* variables $s = (a_i)_{i \in S}$,
4. *state update* variables $s' = (a_i)_{i \in S'}$, and
5. *witness* variables $\omega = (a_i)_{i \in \Omega}$.

We say that the sequence of r variable vectors $((a_i)_{i \in [1,m],1}, (a_i)_{i \in [1,m],2}, \ldots, (a_i)_{i \in [1,m],r}) \in \mathrm{R}^r$ with initial state s'_0 has *state consistency*, if for all $t \in [1, r]$,

we have $s_t = s'_{t-1}$. We call r the number of iterations and include it in pp. In order to relate corresponding state and state update pairs, we require a bijective map $s2s$ between index sets for states $s2s : S \to S'$.

An ADSC-SNARK for R_λ consists of the following tuple of algorithms (Gen, Setup, Auth, Prove, Verify).

Algorithm pp $\leftarrow \mathsf{Gen}(\lambda)$: on input security parameter λ, Gen outputs public parameters pp including a relation $\mathrm{R} \in \mathrm{R}_\lambda$. All other algorithms below use pp as an additional input, so for brevity we do not include pp in their description.

Algorithm $(\sigma_p, \sigma_v, \sigma_a, c_0) \leftarrow \mathsf{Setup}(s'_0)$: takes initial state s'_0 as input and computes a prover-key σ_p, a verifier-key σ_v, and an authentication-key σ_a. It also outputs a commitment to the initial state c_0.

Algorithm $\nu_t \leftarrow \mathsf{Auth}(\sigma_a, t, x_t)$: gets authentication-key σ_a, time step t, and an input x_t and outputs a signature ν_t.

Algorithm $(\pi_t, c_t, p_t) \leftarrow \mathsf{Prove}(\sigma_p, a_t, \nu_t, p_{t-1})$: gets prover-key σ_p, variables $a_t = (\phi_t, x_t, s_t, s'_t, \omega_t)$, and signature ν_t. It outputs a proof π_t and a commitment c_t to state update s'_t. A prover is allowed to have some internal state p_t that is stored between iterative invocations of the proving algorithm.

Algorithm $b \leftarrow \mathsf{Verify}(\sigma_v, \phi_t, \pi_t, c_t, c_{t-1}, t)$: gets verifier-key σ_v, the public input-output ϕ_t, the proof π_t, the current commitment c_t, the commitment from the previous iteration c_{t-1} and timestep t. It either accepts the proof by outputting $b = 1$ or rejects it by outputting $b = 0$.

Discussion: To ease understanding, we briefly explain the above with the help of function f, our running example from the introduction. Relation R represents a computation of function f such that R is satisfied if $(y_t, s'_t) = f(x_t, s_t)$. State is represented by the two same-sized tuples s and s'. The former is part of the input of f, the latter (the state update for that iteration) is part of the output of f. When iteratively applying f on state s, such that $s_{t+1} = s'_t$, we get $y_t, s_{t+1} = f(x_t, s_t)$. ϕ corresponds to the function's output y and private input x corresponds to the function's input x_t for iteration t.

2.2 ADSC-SNARK Properties

For an ADSC-SNARK we require completeness, knowledge-soundness, and succinctness. Note that for the following definitions, the number of iterations $r \in \mathbb{N}$ is part of the public parameters pp.

Intuitively, completeness means that $\mathcal{V}$ accepts all proofs, if in each iteration: 1) relation R is satisfied, 2) the state between consecutive iterations is consistent, and 3) private inputs have been authenticated.

Definition 1 (Completeness of ADSC-SNARK).

For all $\lambda \in \mathbb{N}$, $\mathrm{pp} \leftarrow \mathsf{Gen}(\lambda)$, $(a_1, ..., a_r) \in \mathrm{R}^r$, *with* $a_t = (\phi_t, x_t, s_t, s'_t, \omega_t)$, *and* $s'_0 \in \mathbb{H}^{|S|}$ *such that* $\bigwedge_{t=1}^{r}(s_t = s'_{t-1})$ *and* $p_0 = \{\}$*:*

$$\Pr\left[\begin{array}{c}(\sigma_p, \sigma_v, \sigma_a, c_0) \leftarrow \mathsf{Setup}(s'_0); \\ \nu_t \leftarrow \mathsf{Auth}(\sigma_a, t, (a_{t,i})_{i\in X}) \,\mathrm{for}\, t \in [1, r]\,; \\ (\pi_t, c_t, p_t) \leftarrow \mathsf{Prove}(\sigma_p, a_t, \nu_t, p_{t-1}) \,\mathrm{for}\, t \in [1, r] : \\ \bigwedge_{t=1}^{r} \mathsf{Verify}(\sigma_v, \phi_t, \pi_t, c_t, c_{t-1}, t) = 1\end{array}\right] = 1.$$

With knowledge-soundness, intuitively, for each iteration t in which $\mathcal{V}$ accepts, relation R must be satisfied, the prover must know the witness, and the state between the previous relation and the current relation must be consistent. Additionally, the private inputs must have been authenticated. This must hold, even if an adversary prover can query Auth. We let $(\pi_t, c_t, \phi_t) \leftarrow \mathcal{A}^{\mathsf{Auth}(\cdot)}(\sigma_p, \sigma_v, t, \pi_0)$ be an adversary with oracle access to Auth, that takes public parameters, as well as prover and verifier key as input and computes a proof/commitment/public input-output triple. Then, for knowledge-soundness, we require the existence of probabilistic algorithm $(x_t, s_t, s'_t, \omega_t) \leftarrow \mathsf{Extract}(\sigma_p, \sigma_v, t, \pi_0)$, which gets to see the same random coins as $\mathcal{A}$ and outputs private inputs s_t, state s_t, state-update s'_t and witness ω_t.

Definition 2 (Knowledge-Soundness of ADSC-SNARK). *For all PPT adversaries* $\mathcal{A}$ *with oracle access to* Auth*, there exists a PPT algorithm* $\mathsf{Extract}$*, such that*

$$\Pr\left[\begin{array}{c}\mathrm{pp} \leftarrow \mathsf{Gen}(\lambda); \\ (\sigma_p, \sigma_v, \sigma_a, c_0) \leftarrow \mathsf{Setup}(s'_0); \\ ((\pi_t, c_t, \phi_t); (x_t, s_t, s'_t, \omega_t)) \leftarrow \\ (\mathcal{A}^{\mathsf{Auth}(\cdot)} \,||\, \mathsf{Extract})(\sigma_p, \sigma_v, t, \pi_0) \ \textit{for}\ t \in [1, r] : \\ \bigvee_{t=1}^{r} \left(\begin{array}{c}\left(\begin{array}{c}(\phi_t, x_t, s_t, s'_t, \omega_t) \notin \mathrm{R} \\ \vee s_t \neq s'_{t-1} \\ \vee (t, x_t) \notin \tilde{X}\end{array}\right) \\ \wedge \mathsf{Verify}(\sigma_v, \phi_t, \pi_t, c_t, c_{t-1}, t) = 1\end{array}\right)\end{array}\right] \approx 0,$$

where $\tilde{X}$ *is the set of tuples* $(\tilde{t}, \tilde{x}_t)$ *for each query which* $\mathcal{A}$ *made to* Auth*.*

Intuitively, succinctness implies that the proof and the commitment $(\pi,\ c)$ are asymptotically smaller than the relation (excluding public input-output), and verifying is asymptotically cheaper than checking the relation directly. In our specific situation with ADSC-SNARKs, both proof size $|(\pi, c)|$ and runtime T_{Verify} of Algorithm Verify are even constant in the length of witness ω, the length of input x, the size of states s. The verifier runtime is linear only in the size of statement ϕ. We formalize this with the following definition.

Definition 3 (Succinctness of ADSC-SNARK). *For all* $\lambda \in \mathbb{N}$, $\mathrm{pp} \leftarrow \mathsf{Gen}(\lambda)$, $(a_1, ..., a_r) \in \mathrm{R}^r$, $(\phi_t, x_t, s_t, s'_t, \omega_t) = a_t$, $s'_0 \in \mathbb{H}^{|s|}$, $(\sigma_p, \sigma_v, \sigma_a, c_0) \leftarrow \mathsf{Setup}(s'_0)$, $t \in [1, r]$, $\nu \leftarrow \mathsf{Auth}(\sigma_a, t, (a_{t,i})_{i\in X})$, $p_0 = \{\}$, $(\pi_t, c_t, p_t) \leftarrow \mathsf{Prove}(\sigma_p, a_t, \nu, p_{t-1})$*:*

$$|(\pi_t, c_t)| \in O(1) \ \textit{and}\ T_{\mathsf{Verify}} \in O(|\phi|),$$

i.e., they are constant in $|\omega|, |x|, |s|$*, and* T_{Verify} *is linear in* $|\phi|$*.*

Discussion The definition for knowledge-soundness of ADSC-SNARK is a combination of knowledge-soundness of a regular SNARK in addition to authenticity and state-consistency. The idea is, that an adversary could behave incorrectly in three different ways: First, it could try to generate a proof for a statement, which does not satisfy relation R. Second, it could use a state for the relation, that is not consistent with the state-update from the previous relation. Finally, it could use private inputs, that have not been authenticated by an authorized party. The definition disallows these cases by requiring the probability of any of them occurring to be negligible. For authenticity, the definition captures chosen message attacks against the authenticating party by giving $\mathcal{A}$ oracle access to Auth.

3 Main ADSC-SNARK Construction

Before presenting our main contribution, the construction of ADSC-SNARK, we start with a more formal overview and highlight its key concepts.

3.1 Overview

Our ADSC-SNARK takes the SNARK of Groth [40] as a starting point. There, a proof comprises 3 group elements $[A]_1, [B]_2$, and $[C]_1$ that the prover derives by computing linear combinations of elements from its prover key σ_p. Informally, the security rationale is that the prover cannot create a valid proof besides by linearly combining elements from σ_p, as the verifier will check certain relations between the proof elements that depend on a set of secret values (α, β, $\gamma, \ldots$) drawn randomly during setup. Note that the verifier does not need to see the secret values in the clear, as they can perform verification with the help of pairing e. As the prover does not know the secret values either, they can create a verifying proof only by a (linear) combination of the prover key elements that depend on the secret values, but do not reveal them. In fact, the prover must use variables $(\phi, x, s, s', \omega) \in \mathrm{R}$ as coefficients of these linear combinations for a proof to verify. Recall that $(\phi, x, s, s', \omega) \in \mathrm{R}$ implies that the computation was carried out correctly.

Our key idea to extend the Groth SNARK such that it also provides state-consistency and data authenticity is loosely based on the observation made and used in several other works [17, 24, 32, 46]: proof elements not only allow the verifier to check the satisfaction of R, but can also act as a *commitment* to the variables satisfying R. Intuitively, if you assume that the involved prover key elements are random, any two different linear combinations of them are also different with high probability. This is similar to the standard Pedersen commitment [53]. Using the commitment property allows for checking between two different SNARK proofs whether they were carried out on the same (committed)

data. Also, it allows to verify whether the data used in a SNARK proof complies with the data commited to.

State Consistency: By defining state variables s and s' as the committed data, we can use the above technique to achieve state-consistency. More specifically, the prover outputs a proof π_{t-1} and a vector commitment c_{t-1} in step $t-1$. c_{t-1} commits to the state update s'_{t-1}. In the subsequent iteration t, the prover produces another proof π_t (and also another commitment c_t). Now, in this iteration t, the verifier can not only check whether π_t is correct, but also whether state s_t committed to by some of the elements in π_t is the same as s'_{t-1} committed to by c_{t-1}. In our ADSC-SNARK construction, we denote c_t by element $[E_t]_1$.

Authenticated Data: For proving data authenticity, we let element $[D_t]_1$ serve as commitment for private input x. First, the authenticating party signs the commitment with a regular signature scheme. Now the verifier can check whether the private input variables used for π_t match the committed input variables used for $[D_t]_1$. By also checking the signature on $[D_t]_1$ the verifier checks that the private input stems from the authenticating party.

Method: In ADSC-SNARKs, we design a novel strategy for checking proofs and commitments at the same time: To verify, that commitments $[D_t]_1$, $[E_t]_1$, $[E_{t-1}]_1$ and proof π_t have been computed using the same data, we let Setup generate random group elements $R_i, T_i \in \mathbb{G}_1$. We then subtract them from prover key elements $(\left\{\frac{P_i(z)}{\delta}\right\}_{i\in S}, \left\{\frac{P_i(z)}{\delta}\right\}_{i\in S'}, \left\{\frac{P_i(z)}{\delta}\right\}_{i\in X})$ that correspond to the variables for the state s_t, the state update s'_t, and private input x. This forces the prover to provide commitments E_{t-1}, E_t, and D_t with the same coefficients as used for the modified prover key elements $\left\{\frac{P_i(z)-R_{s2s(i)}}{\delta}\right\}_{i\in S}$ $\left\{\frac{P_i(z)-R_i}{\delta}\right\}_{i\in S'}$, $\left\{\frac{P_i(z)-T_i}{\delta}\right\}_{i\in X}$). Commitments E_{t-1}, E_t, and D_t will be added by the verifier to the verification equation. If the prover provided the commitments correctly, the additional linear combinations of T_i and R_i will cancel out. We use additional random secrets $\eta, \kappa, \varepsilon$ to prevent the prover from mixing coefficients across the different commitments. This is different from the approaches by Costello et al. [24] and Campanelli et al. [17], which do not subtract random group elements from the prover key, but require a special non-degeneracy condition for Relation R. The latter approach leads to additional arithmetization overhead, which is avoided in our construction.

Summary: In conclusion, our ADSC-SNARK is a Groth16 SNARK with the following modifications:

- Setup generates additional random $\mathbb{G}_1$ elements T_i, R_i and secret $\mathbb{F}_p^*$ elements $\eta, \kappa, \varepsilon$.
- Setup subtracts ηR_i, κR_i, and εT_i from the prover key elements corresponding to S, S', and X.
- Auth computes vector commitment $[D_t]_1$ on the private input data x_t and a signature sig of $[D_t]_1$.

– Prove produces commitments $[E_t]_1$ for state-consistency.
– The verifier includes commitments $[E_{t-1}]_1, [E_t]_1$, and $[D_t]_1$ in the verification equation.
– The verifier checks signature sig on $[D_t]_1$.

3.2 ADSC-SNARK Relation

Relation R in our ADSC-SNARK construction will be given as a Quadratic-Arithmetic-Program (QAP) [36]. QAPs have turned out to be useful in practice, as there exist straight-forward reductions from Arithmetic Circuits (see [52]) and from Rank-1-Constraint-Systems (R1CS), a popular type of relation to represent computations [6].

We define a QAP over the following parameters: a field $\mathbb{H}$, degree d, number of variables m, and a partition of the variables with index sets Φ, X, S, S', Ω, such that $|S| = |S'|$ and $\Phi \cup X \cup S \cup S' \cup \Omega = [1, m]$. A QAP over these parameters is the sequence of polynomials $\{u_i(Z)\}_{i\in[0,m]}$, $\{v_i(Z)\}_{i\in[0,m]}$, $\{w_i(Z)\}_{i\in[0,m]}$, $t(Z)$, such that $u_i(Z), v_i(Z), w_i(Z)$ have degree equal or less than $d-1$ and $t(Z)$ has degree d.

The QAP is satisfied for variables $a_i \in \mathbb{H}$ and constant $a_0 = 1$, if there exists a polynomial $h(Z)$, such that

$$\sum_{i\in[0,m]} a_i u_i(Z) \cdot \sum_{i\in[0,m]} a_i v_i(Z) - \sum_{i\in[0,m]} a_i w_i(Z) = h(Z)t(Z). \tag{1}$$

Setting $a_0 = 1$ conveniently enables reductions from R1CS or Arithmetic Circuits with constant terms to QAPs.

Discussion Many of the constructions by related works ([3, 17, 24]) require a special non-degeneracy condition for the QAP, which is that the polynomials for $i \in S \cup S'$ or $i \in X$ are linearly independent. This is usually achieved by augmenting the relation by dummy constraints (see e.g. [51]), which increase degree d of the QAP by the size of the set of indices for the independent polynomials, i.e. $|S| + |S'|$ or $|X|$. This increases prover time and is accounted for as arithmetization overhead in Table 1. Our construction specifically avoids this requirement.

3.3 Technical details

In ADSC-SNARKs, public parameters pp = (QAP, $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, e, $\mathbb{F}_p$, r) comprise the tuple of relation R = QAP, the descriptions of a pair of groups $\mathbb{G}_1$, $\mathbb{G}_2$ with a bilinear map e and target group $\mathbb{G}_T$, and a prime field $\mathbb{H} = \mathbb{F}_p$ such that $|\mathbb{F}_p| = p = |\mathbb{G}_1|$ and $|p| = \lambda$. Parameters pp also include the number of iterations r. Also, ADSC-SNARKs employ a standard digital signature scheme (SigGen, SigSign, SigVerify) with

– $(sk, pk) \leftarrow \mathsf{SigGen}(\lambda)$ generates private key sk and public key pk,

$(\sigma_p, \sigma_v, \sigma_a, c_0) \leftarrow \mathsf{Setup}(s'_0)$

$T_i \leftarrow\!\$\ \mathbb{F}_p^* \text{ for } i \in X, \quad R_i \leftarrow\!\$\ \mathbb{F}_p^* \text{ for } i \in S', \quad \alpha, \beta, \gamma, \delta, \kappa, \eta, \varepsilon, z \leftarrow\!\$\ \mathbb{F}_p^*,$

$(sk, pk) \leftarrow \mathsf{SigGen}()$

$$\sigma_{p,1} = \left(\alpha, \{u_i(z)\}_{i\in[0,m]}, \left\{\frac{P_i(z)}{\delta}\right\}_{i\in\Omega}, \left\{\frac{P_i(z)-\eta R_{s2s(i)}}{\delta}\right\}_{i\in S}, \left\{\frac{P_i(z)-\kappa R_i}{\delta}\right\}_{i\in S'} \right.$$
$$\left. \left\{\frac{P_i(z)-\varepsilon T_i}{\delta}\right\}_{i\in X}, \left\{\frac{z^i t(z)}{\delta}\right\}_{i\in[0,d-2]}, \{R_i\}_{i\in S'}\right)$$

$$\sigma_{p,2} = \left(\beta, \{v_i(z)\}_{i\in[0,m]}\right), \sigma_{v,1} = \left(\left\{\frac{P_i(z)}{\gamma}\right\}_{i\in\{0\}\cup\Phi}\right), \sigma_{v,2} = \left(\gamma, \delta, \varepsilon, \eta, \kappa\right).$$

$$\sigma_p = \left([\sigma_{p,1}]_1, [\sigma_{p,2}]_2\right), \sigma_v = \left([\sigma_{v,1}]_1, [\sigma_{v,2}]_2, [\alpha\beta]_T, pk\right), \sigma_a = (sk, \{[T_i]_1\}_{i\in X}),$$

$$c_0 = [E_0]_1 = \sum_{i\in S'} s'_{0,\mathrm{pos}_{S'}(i)} [R_i]_1.$$

$\nu_t \leftarrow \mathsf{Auth}(\sigma_a, t, x_t)$

$$D = \sum_{i\in X} a_i T_i, \quad sig = \mathsf{SigSign}(sk, ([D]_1, t)), \quad \nu_t = ([D]_1, sig).$$

$(\pi_t, c_t) \leftarrow \mathsf{Prove}(\sigma_p, a_t, \nu_t)$

$$h(Z) = \left(\sum_{i\in[0,m]} a_{t,i} u_i(Z) \cdot \sum_{i\in[0,m]} a_{t,i} v_i(Z) - \sum_{i\in[0,m]} a_{t,i} w_i(Z)\right) / t(Z).$$

$$A = \alpha + \sum_{i\in[0,m]} a_i u_i(z), \quad B = \beta + \sum_{i\in[0,m]} a_i v_i(z), \quad E = \sum_{i\in S'} a_i R_i,$$

$$C = \sum_{i\in\Omega} \frac{a_i P_i(z)}{\delta} + \sum_{i\in S} \frac{a_i \left(P_i(z) - \eta R_{s2s(i)}\right)}{\delta} + \sum_{i\in S'} \frac{a_i \left(P_i(z) - \kappa R_i\right)}{\delta}$$
$$+ \sum_{i\in X} \frac{a_i \left(P_i(z) - \varepsilon T_i\right)}{\delta} + \frac{h(z)t(z)}{\delta},$$

$$\pi_t = \left([A]_1, [B]_2, [C]_1, [D]_1, sig\right), \quad c_t = [E]_1.$$

$v \leftarrow \mathsf{Verify}(\sigma_v, \phi_t, \pi, c_t, c_{t-1}, t)$

$$\text{Accept, iff } [A_t]_1 \cdot [B_t]_2 = [\alpha\beta]_T + \left(\sum_{i\in\{0\}\cup\Phi} a_{t,i} \left[\frac{P_i(z)}{\gamma}\right]_1\right) \cdot [\gamma]_2 + [C_t]_1 \cdot [\delta]_2$$
$$+ [D_t]_1 \cdot [\varepsilon]_2 + [E_{t-1}]_1 \cdot [\eta]_2 + [E_t]_1 \cdot [\kappa]_2$$
$$\text{and} \quad \mathsf{SigVerify}(pk, sig_t, ([D_t]_1, t)) = 1.$$

Fig. 1. ADSC-SNARK construction (without zero-knowledge). The differences to the Groth16 SNARK are highlighted.

– $sig \leftarrow \mathsf{SigSign}(sk, m)$ computes a signature sig on a message m using the private key,
– and $b \leftarrow \mathsf{SigVerify}(pk, sig, m)$ verifies signature sig against message m using public key pk and outputs a verification bit b.

Figure 1 shows the protocol of our ADSC-SNARK. The differences to the Groth16 SNARK are highlighted. Here, we ignore prover state p_t, as it will appear only in the zero-knowledge version of our ADSC-SNARKs. To improve readability, we write $P_i(z)$ for $\beta u_i(z) + \alpha v_i(z) + w_i(z)$. Recall that function $s2s$ maps indices from states S to state updates S'. This means that if $i \in S$, then a_i is a state and $a_{s2s(i)}$ is its update.

3.4 Security Analysis and Zero-Knowledge

We defer the detailed security analysis to Appendix A. Similarly, the zero-knowledge variation of our main construction is presented in Appendix B.

3.5 Optimizations

The construction above contains a series of optimizations that might be of independent interest.

– For the setting without zero-knowledge, the randomization factors originally used by Groth [40] can be set to zero, which eliminates the terms with A and B in the equation for proof element C. Therefore, the terms for B do not need to be calculated both in groups $\mathbb{G}_1$ and $\mathbb{G}_2$, but only in $\mathbb{G}_1$, reducing prover time.
– The prover-key contains direct evaluations of the polynomials $u_i(z)$ and $v_i(z)$. This is suggested in the paper by Groth [40], but the presented construction there shows a version where instead the prover-key contains monomials z^i. The latter variant requires the prover to do additional Fast-Fourier-Transformations (FFTs) to transform coefficients of u_i and v_i to the monomial basis.
– Subtracting the randomly drawn elements $[T_i]_1$ and $[R_i]_1$ for a commitment c from the corresponding prover key elements $[P_i(z)/\delta]_1$ in σ_p, and letting the verifier add c directly to the verification equation is a new efficient method to check, whether commmitted variables for commitment c match the variables for proof π. It does not require an additional strengthening of the QAP and does not require additional intermediate proof elements.

3.6 Extension: Multiple Authenticators

It is possible to have several independent authenticating parties J which each can only authenticate a subset X_j of the inputs, without the need of the authenticating parties to communicate with each other. In order to ensure that inputs

have been provided by the correct authentication party,
$\mathsf{Setup_a}$ produces multiple key pairs for the signature scheme:

$$(sk_j, pk_j) \leftarrow \mathsf{SigGen}() \text{ for } j \in J$$

and distributes authentication key $\sigma_{a,j} = (sk_j, \{T_i\}_{i \in X_j})$ to authenticating party j.
Each authenticating party authenticates its input subset

$$\nu_{j,t} \leftarrow \mathsf{Auth}(\sigma_{a,j}, t, x_{j,t})$$

and sends it to the prover.
The prover includes all $[D_j]_1, sig_j$ from $\nu_{j,t}$ in proof π_t.
The verifier checks the signatures of every $[D_j]_1$:

$$\mathsf{SigVerify}(pk, sig_{j,t}, ([D_{j,t}]_1, t)) = 1.$$

It computes $[D_t]_1 = \sum_{j \in J} [D_{j,t}]_1$ and uses it to check the original verification equation. It rejects the proof, if any of the checks fails and accepts otherwise.

Above extension will increase the proof size by $|J| - 1$ signatures and $\mathbb{G}_1$ elements. It will also increase verification time by the additional $|J| - 1$ signature verifications.

4 Evaluation

We compare the performance of the full zero-knowledge version of our ADSC-SNARK against the folklore ("strawman") approach where a signature verification algorithm and a hash function are arithmetized for a regular SNARK. Additionally, we compare our ADSC-SNARK with two more recent, specialized SNARKs. First, we compare it with the AD-SNARK by Backes et al. which allows proofs for computations on authenticated data [3] and, second, a construction that we name *LegoGro SC-SNARK*. The latter is a composition of two LegoGro16 SNARKs from the LegoSNARK framework [17] achieving state-consistency through functional composition.

We stress that neither AD-SNARKs (Backes et al.) nor LegoGro SC-SNARKs provide the same properties as the full ADSC-SNARK construction. They *either* provide authentication *or* state consistency, but not both at the same time.

4.1 Implementation Details

We have implemented our ADSC-SNARK, the Strawman ADSC-SNARK and the LegoGro SC-SNARK in the popular libsnark C++ library[2]. We instantiate the signature scheme by EdDSA on an Edwards curve of similar prime order group size as the SNARK curve.

[2] https://github.com/scipr-lab/libsnark

The **strawman ADSC-SNARK** for relation R is a Groth16 SNARK for an augmented relation $\mathrm{R}_a = \mathrm{R} \wedge \mathrm{R}_x \wedge \mathrm{R}_s \wedge \mathrm{R}_{s'} \wedge \mathrm{R}_i$. Here, R_x encodes a signature verification for the private input x. Relations R_s and $\mathrm{R}_{s'}$ encode collision resistant hash functions applied to state s and state-update s'. Finally, R_i encodes a counter-increment preventing that signed private input can be reused across iterations. The hash outputs are made public input-output variables of the Groth16 SNARK and are considered part of the proof of the strawman ADSC-SNARK. State consistency can be checked in the verification algorithm by checking equality between corresponding state hashes additionally to the SNARK verification equation.

While integrating R_x and R_s into SNARKs have previously been found to be expensive (e.g., Backes et al. assume at least 1000 constraints per private input for a signature verification arithmetization [3]), recent progress has led to significant improvements, and our implementations build on these improvements for fair comparisons. Specifically, for the signature verification, we have modified the already efficient EdDSA verification gadget in the open-source gadget library ethsnarks[3] by instantiating it with the SNARK-friendly Poseidon hash described by Grassi et al. [39]. We have further reduced the constraint count by fixing the public key before arithmetization. Elliptic curve operations are efficiently realized on a SNARK-friendly curve with curve points represented in affine montgomery coordinates. Scalar multiplication uses fewer constraints due to 3 bit lookup tables. This results in a constant number of 2126 constraints per signature and additionally 72 constraints per private input. We assume that there is a single signature for all inputs when the strawman approach is taken as the baseline performance. This is the most conservative assumption, so speedups will likely be higher in practice than the reported numbers. Note that, in contrast to this, for our ADSC-SNARK, there is no difference in prover performance whether the message authentication codes have been produced with the same private authentication key, or if many private authentication keys have been used.

For R_s and $\mathrm{R}_{s'}$, we use the same Poseidon hash as in R_x, resulting in 144 constraints per state variable in total, as there are two hash functions for each state. Relation R_i adds one plain constraint and one state variable, yielding 145 additional constraints in total.

Backes et al. provide a libsnark implementation of their **AD-SNARK** construction, supporting both a designated and a public verfier version. We benchmark against both versions.

For **LegoGro SC-SNARK**, the LegoSNARK framework by Campanelli et al. [17] enables the composition of proofs including function composition. That is, for functions $y = f(x), z = h(y)$ and SNARK algorithms $SNARK_f, SNARK_h$, the framework provides a compiler to construct a new SNARK from $SNARK_f$ and $SNARK_h$ to create proofs for the relation representing $z = h(f(x))$. For our setup, we can view the iterative application of some function f on state s as a function composition of f with itself and therefore use the LegoSNARK framework to achieve state-consistency. For that, we use the LegoGro16 prim-

[3] https://github.com/HarryR/ethsnarks

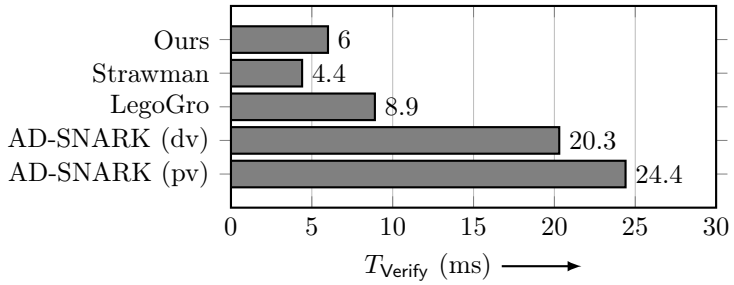

Fig. 2. Verifier runtimes of our zero-knowledge ADSC-SNARK, strawman SNARK, LegoGro SC-SNARK and AD-SNARK designated verifier (dv) and public verifier (pv). AD-SNARK is evaluated for $|X| = 1$ private inputs. LegoGro SC-SNARK and AD-SNARK do not provide both state consistency and authenticated input data.

itive described by Campanelli et al.. We define two index sets for state s and state-update variables s' which is equivalent to two subdomains in their paper. We implemented LegoGro16 in libsnark and apply their composition scheme from their Theorem 3.1 recursively to get LegoGro SC-SNARK.

4.2 Benchmarks

We have run the benchmarks with the BN-254 elliptic curve which is a standard choice for pairing-based SNARKs. We have compiled the code with the GCC C++ compiler on optimization level 3 and ran benchmarks on an i5-3570K Intel Core Processor with a Ubuntu 24.04 operating-system and 24 GB RAM. We ran the benchmarks within a single thread for better comparability. For each data point, we have constructed a basic relation with 2^{15} constraints and up to $|X| = 2^{20}$ private inputs and $|S| = 2^{20}$ states. We evaluated the Strawman SNARK for up to $|X| = 2^{14}$ private inputs and $|S| = 2^{14}$ states as for a larger number of inputs and states, our 24 GByte RAM machine runs out of memory when running the Strawman SNARK. The number of public input-output variables was set to 0.

Note: We stress that we compare ADSC-SNARKs with LegoGro SC-SNARKs and AD-SNARKs only to put the performance of ADSC-SNARKs into perspective. Neither LegoGro SC-SNARKs nor AD-SNARKs provide the same properties as ADSC-SNARKs, see Table 1.

Figure 2 compares T_{Verify} of the different SNARKs. The verifier runtimes of our ADSC-SNARK, the strawman SNARK, and LegoGro SC-SNARK are constant. For AD-SNARK, the plot shows the runtime with $|X| = 1$ private inputs. Compared to AD-SNARK, our SNARK verifier needs 70 % (designated verifier) and 75 % (public verifier) less runtime. However, as their verifier iterates over all private input labels, the AD-SNARK verifier is asymptotically linear with respect to the number of private inputs and therefore will increase in settings with many inputs. This is even more significant for the public verifier version,

where the verifier needs to compute a linear number of pairing evaluations with respect to $|X|$ (see Table 1 for details). For $|X| = 2^{15}$ private inputs, T_{Verify} is 101.4 ms for the designated verifier version and 34.9 s for the public verifier version. The strawman SNARK has the fastest verifier runtime as it invokes the unmodified Groth16 verifier with a hash digest as a public input. Our ADSC-SNARK is faster than the LegoGro SC-SNARK (33 % less runtime), because the LegoGro16 verifier has additional overhead to allow for state-consistency between different proof systems, while our construction is optimized to the setting of a single proof system and a single relation. The AD-SNARK verifier is the slowest, even with the designated verifier version and just one input.

Table 2. Proof size for curve BN254

System	Elements	Size
Strawman SNARK	$2 \times \mathbb{G}_1, 1 \times \mathbb{G}_2, 1 \times \mathbb{F}_r$	166 B
LegoGro SC-SNARK	$5 \times \mathbb{G}_1, 1 \times \mathbb{G}_2$	236 B
AD-SNARK (dv)	$8 \times \mathbb{G}_1, 3 \times \mathbb{G}_2$	406 B
ADSC-SNARK	$5 \times \mathbb{G}_1, 1 \times \mathbb{G}_2, 1 \times sig$	271 B

Table 2 presents the proof size for each proof system. The center column shows the elements of each proof, the right column shows the actual size after serialization when using the BN254 curve. The serialization routines compress elliptic curve points by storing just one coordinate and a sign bit. However, there is additional overhead for paddings to align objects to 8-bit data chunks and for additional separation tokens between data objects. While the strawman SNARK achieves the smallest proofs, our SNARKs and LegoGro SC-SNARK are still smaller than the designated verifier AD-SNARK. The proof of the public verifier AD-SNARK has $|X|$ additional elements from $\mathbb{G}_2$ as well as $|X|$ signatures. The

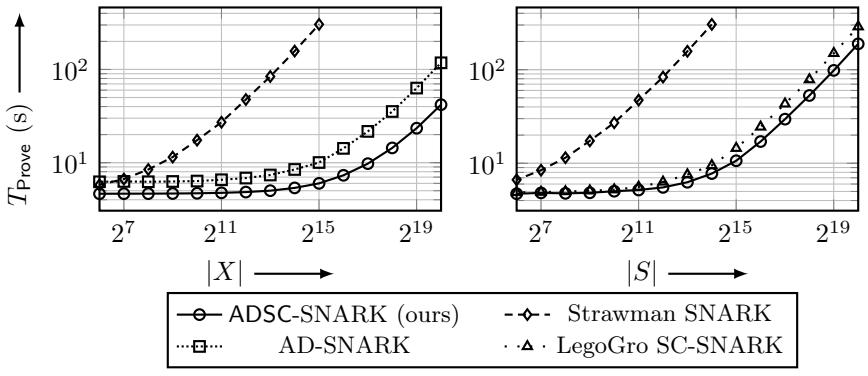

Fig. 3. Prover runtime. Left: varying private inputs. Right: varying number of states.

left plot in Figure 3 compares the prover runtime with respect to the number of private inputs. The prover runtime for the designated and public AD-SNARK verifier is the same. For 2^{14} inputs, our ADSC-SNARK is 29× faster than the strawman SNARK and 1.6× faster than AD-SNARK. Note, that AD-SNARK does not support state-consistency. The right plot compares the prover runtime with respect to the number of states. For 2^{14} states, our zero-knowledge ADSC-SNARK is 39× faster than the strawman SNARK and 1.2× faster than LegoGro SC-SNARK.

Table 3. Prover runtime for strawman SNARK vs. ADSC-SNARK and speedup factor for varying states and inputs for a small relation with 2^{10} constraints. $|X|$: number of inputs, $|S|$: number of states.

$\|S\|$	2^0			2^7			2^{14}		
$\|X\|$	Strawman	Ours	Speedup	Strawman	Ours	Speedup	Strawman	Ours	Speedup
2^0	0.64 s	0.25 s	×2.5	3.80 s	0.32 s	×11.9	303 s	4.38 s	×69
2^7	2.46 s	0.27 s	×9.1	5.05 s	0.33 s	×15	303 s	4.30 s	×70
2^{14}	154 s	1.04 s	×147	155 s	1.11 s	×140	447 s	5.03 s	×89

Table 3 shows the prover runtimes of the strawman SNARK and our zero-knowledge ADSC-SNARK as well as the ratio between the runtimes with respect to both the number states and private inputs. The number of private inputs and states is at most 2^{14} due to the heavy memory requirements of the strawman SNARK. To set the effects of the input and state size on the prover runtime in relation to the constraint size of the base relation, we chose a smaller relation with 2^{10} constraints. We do not include AD-SNARK or LegoGro SC-SNARK, as they only support either states or private inputs but not both. The table shows a significant speedup in prover time that increases with the number of states and inputs, for 2^{14} inputs and states, it faster by a factor of ×89.

Summarizing, our construction scales well with the number of private inputs and states and has good concrete prover performance. It is significantly faster than the strawman approach and even outperforms AD-SNARK and LegoGro SC-SNARK.

5 Applications

ADSC-SNARKs have been designed for the use in digital control systems. A typical digital control system comprises sensors that measure some state of the system to be controlled, a control unit that uses sensor measurements to compute control outputs, and actuators that manipulate the system accordingly. In most cases, control units have an internal state that is periodically updated and compute outputs at a fixed frequency. Therefore, the typical computation of a digital control unit is stateful and applied to many inputs.

Prominent examples are flight control in aircraft, electrical power grid control, power generation plant control or control in automated industrial processes. Many of these systems are safety critical, i.e., a malfunction or adversarial tampering can lead to loss of lives or cause severe economic consequences. In order to meet rising demands for efficiency, autonomy, and adaptability of such systems, their complexity and interconnectivity increases. This results in new attack surfaces for malicious actors and also introduces new sources for non-malicious but unintended system failures due to design errors or random faults in components. The discovery of sophisticated cyber-attacks on industrial control systems, including Stuxnet [29] or Duqu [21], demonstrates the importance of securing these systems.

Applying ADSC-SNARK in such a setting is beneficial for several reasons. First, SNARKs could be used as a security measure to prevent cyber-attacks on control units or communication channels. The idea would be to require the commands from a control unit to carry a proof about correct computation on authenticated inputs. Second, in highly critical areas such as aviation, hardware failures of control units can lead to complex and unpredictable failure behaviour which must be mitigated [26]. Current solutions to detect these failures require using redundant computers and comparing their outputs. SNARKs could be used as an alternative for detecting malfunctions reducing the number of redundant components [54]. We further discuss the application of ADSC-SNARK for securing a power grid control system and for safety in a flight control system.

5.1 Power Grid Control

A power grid enables the distribution of electric power from producers such as power plants, wind- or solar farms to power consumers, such as factories or individual households. Power grid control ensures an equilibrium between power demand and power production. In this setting, sensors measure loads and AC-frequency at various locations in the grid. Power producers act as the control system's actuators. A control center (the control unit) commands producers to increase or decrease power production depending on the demand of consumers and current loads on the transmission lines [4]. For example, the Polish power grid has 114 generators that are controlled automatically and in real-time by a central control unit over a wide area network spanning the entire country and connecting to neighboring states [41]. For these systems, many local and remote access points exist, creating a large attack surface [28]. SNARKs could be beneficial in this setting. The idea would be to require commands from the control unit to carry a proof which asserts that the commands are the output of the correct global control function that stabilizes the power grid. In case of an invalid proof, a power producer could request commands from a backup control unit or enter a fail-safe mode. This measure would limit the capabilities of an attacker, even if the attacker had full authority over the central control unit.

5.2 Flight Control System

Fly-by-wire control is standard equipment of modern large aircraft. It is realized by a flight control system consisting of a set of redundant control units (flight control computers), actuators manipulating various control surfaces (rudders, elevators, ailerons) and sensors measuring flight conditions (aircraft velocity and attitude) and pilot commands.

A malfunction of the flight control system is considered catastrophic, as it can result in uncontrollability and therefore loss of the airplane. As a result, reliable fault detection and mitigation techniques need to be employed. The currently prevalent method in safety-critical avionics to ensure computational integrity is replicating control computers, which compare each other's outputs to detect hardware faults [57, 61]. Replicating computers for ensuring computational integrity results in more weight for electronics and cabling and in high development cost. For example, the flight control system of Airbus A340 consists of 5 duplex (10 in total) flight control computers [57], and the flight control system of Boeing 777 consists of 3 triplex flight control computers (9 in total) [61].

We suggest to employ ADSC-SNARKs and demonstrate their practicality in a flight control system, not to protect against cyber-attacks, but against hardware faults. As a consequence, the number of required redundant computers and thus total cost can be reduced. Note that ADSC-SNARKs cannot reduce the number of computers to one, as there still need to be redundant units to meet hardware reliability requirements. For this application, we consider a faulty computer as part of the flight control system to behave as the malicious adversary. With ADSC-SNARK, commands from such a possibly faulty control computer can be verified before being accepted by actuators. Considering a faulty control computer to behave like a malicious "intelligent" adversary might seem unnecessarily conservative at first. However, this is a standard assumption in the analysis of safety-critical avionics systems, as any stronger assumption would require reliable evidence (such as empirical data), which in practice is unfeasible to obtain for the highest safety targets [45]. Indeed, many allegedly unplausible failure behaviors have been observed in operation [26]. Note, however, that cryptographic methods have not yet been employed in practice as a means to protect against hardware faults. To further illustrate this concept, we arithmetized the computation of a flight control law inspired by the flight control of the Airbus A320 aircraft [23, 30, 31] and simulate the generation and verification of ADSC-SNARK proofs thereof.

Simulation Our scenario, as depicted in Fig. 4 consists of four sensor units: Two air data and inertial reference units (ADIRUs) that measure airflow (velocity V, angle-of-attack α, side-slip angle β), attitude (roll angle Φ, pitch angle Θ), rotation (roll rate p, pitch rate q, yaw rate r) and vertical acceleration (n), as well as two transducer units that measure the position of the pilot's controls, which is a sidestick for pitch (q_c) and roll (p_c) commands, and the pedals for sideslip commands β_c. In each time-step, every sensor unit authenticates their measurements and sends them to the central flight control computer. The flight

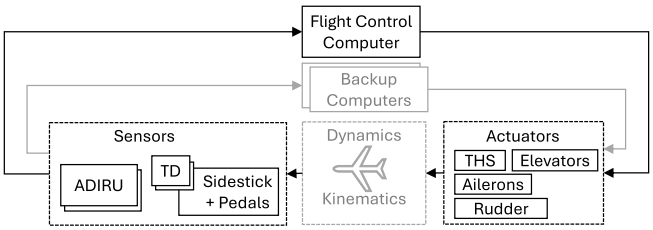

Fig. 4. Flight-control scenario depicting signal flows between sensors, control unit and actuators.

control computer evaluates a stateful flight control law, taking the measurements from the sensors as inputs. The resulting control commands are sent to the actuators for elevators (elevator deflection Δq), ailerons (aileron deflection Δp), rudder (rudder deflection Δr) and trimmable horizontal stabilizer (THS, pitch trim $\Delta\tau$). They also produce a proof-commitment pair (π, c) for each iteration that allow the actuator units to verify the correctness of the commands. We do not simulate the dynamics and kinematics of the aircraft. Also, we do not simulate an additional backup flight control computer, which would be required for increased availability, once a failure of the first control computer has been detected.

Discussion For this setting, the succinctness properties of ADSC-SNARK are particularly important. While the flight control computer can be expected to have a decent amount of computational power and access to all the required input data from various sensors and other systems (either trough connections via a central aircraft data network or a multitude of several field busses), the actuator electronics can be expected to be fairly computationally weak and to have low bandwidth data connections. Therefore, a solution where verification time and proof size increase with the size of the state, the (private) input data or the size of the control law, would be unsuitable, when the number of inputs, states and the size of the law are large. This is why in our solution, verification time and proof size are constant in these metrics. However, the concrete computational cost for verification is still challenging, due to the costly pairing evaluations. Earlier works demonstrated feasibility of computing pairings on a 300 MHz automotive-grade microcontroller, where a pairing on a 158-bit Barreto-Naehring curve was reportedly evaluated in 40.9 ms, requiring 27 kB of flash program memory [2].

Control Law The specific control law evaluated and proven by the control unit is shown as a block diagram in Figure 5. It contains major elements of the Airbus A320 normal flight control law described in references [23, 30, 31] and the A320 flight crew operating manual: Lon and Lat Control are the basic control laws for the longitudinal and lateral aircraft motions. Lon Control integrates the error between pitch command and measured pitch for stationary accuracy and mixes the measurement signals corresponding to the pitch movement (n, q). Lat Con-

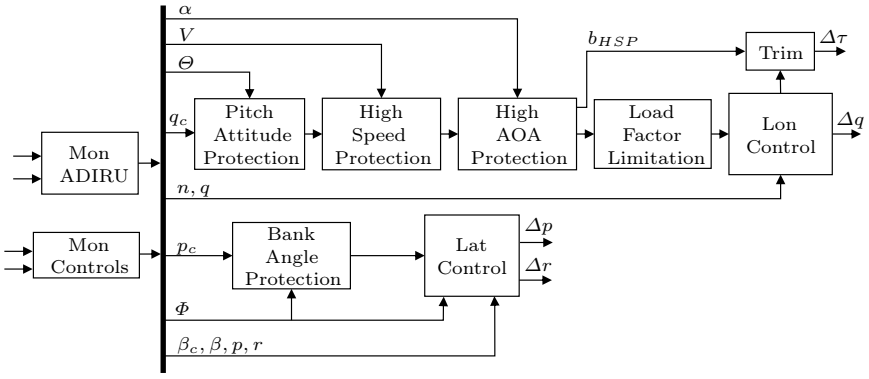

Fig. 5. Block diagram of demo flight control law inspired by Airbus A320 flight control.

trol integrates the roll command and mixes the signals corresponding to the roll and yaw movements (p, r, β, Φ). The Trim block integrates the pitch command resulting in a THS deflection command which results in a steady pitch moment. Due to the integrators, all three blocks are stateful. The pitch attitude protection limits the pilots control inputs, if the pitch angle exceeds 30° nose up or 15° nose down. The high-speed protection adds a proportional pitch-up command to reduce aircraft speed close to the structural speed limits. A hysteresis (activation of the protection is at higher speeds than deactivation) makes this block stateful. The high angle of attack protection changes the pilot input command from a vertical acceleration command to a direct angle-of-attack command in case of a high angle-of-attack. It also limits the pitch command to prevent a stall condition. It is only deactivated when the pilot pushes the stick by more than 8° pitch down or by 0.5° for more than 0.5 s. This condition makes this block also stateful. The load factor limitation limits the pitch command to prevent maneuvers that would exceed the structural limits of the aircraft. Finally, the bank angle protection changes a roll rate command from the side stick to a bank angle command, if the aircraft exceeds a bank angle of 33° and limits the bank angle to at most 67°. We do not claim that this represents the complete flight control function of a commercial transport aircraft, in fact, there are various additional modes for other operational conditions, such as a ground mode, flare mode, abnormal attitude laws and alternate laws with reduced protections. We also do not consider spoiler, engine, slat and flap commands. We use fixed constants for control gains that do not reflect the actual gains, which are not publicly available.

Evaluation Each physical quantitiy is represented as a finite field element, resulting in $|X| = 24$ private input variables (2×9 signals from ADIRU $+2 \times 3$ pilot control signals), $|S| = 6$ states (3 discrete integrators, 3 states for hysteresis in protection blocks), $|\Phi| = 4$ public input-output variables and $|\Omega| = 1506$ witness variables. The resulting QAP is of degree $d = 1593$.

Table 4. Runtime and communication of digital control system demo. Proof size includes proof and commitment.

Elliptic Curve	Control Unit runtime	Actuation System runtime	Proof size
BN254	102 ms	8.9 ms	568 B
BN183	50.4 ms	4.3 ms	432 B
BN124	41.4 ms	3.7 ms	296 B
GMV181	39.6 ms	3.8 ms	448 B
GMV97	29.6 ms	2.6 ms	304 B
GMV58	17.3 ms	1.3 ms	160 B

We have measured I) the runtime for the control unit to process the input data and to create a proof for a valid computation, and II) for the actuation units to verify the commands from the control unit. This includes serializing, writing, reading and deserializing exchanged data in addition to executing the ADSC-SNARK algorithms. Additionally, the size of the proof including commitment is reported.

For sake of simplicity, the simulation does not run on different computers sending the data over a network, but instead runs the logic of the sensors, control unit and actuation system on the same machine.

The numbers reported include the walltime time passed including reading and writing in- and output, not just the processor time as in Section 4. It does not include one-time setup and initialization computations, such as deserializing and preprocessing the verifier-key. The reported proof size includes overhead such as separation characters used in the serialization routines

As the untrusted control computer has limited computational resources, one may significantly reduce the security parameter of SNARKs [54]. Consequently, we present benchmarks for a choice of different elliptic curves and security parameters:

- BN254, BN183, BN124: Barreto-Naehring elliptic curves [5] with embedding degree 12 and 254, 183, and 124 bits prime group order,
- GMV181, GMV97, GMV58: Galbraith-McKee-Valença elliptic curves [35] with embedding degree 6 and 181, 97, and 58 bits prime group order.

Curve BN254 was constructed by Ben-Sasson et al. [10] and is the default curve in libsnark, also referred to as `alt_bn128`. GMV181 is the curve used by Ben-Sasson et al. [8]. We constructed and implemented the other curves according to the methods of Barreto and Naehrig as well as Galbraith et al.. The curve parameters are listed in Appendix C.

Table 4 shows runtime measurements and the communication overhead for sensors (authentication tag size) and control unit (proof size). The simulation has been run with the same settings as the benchmarks in Section 4. The code

is compiled with GCC in optimization level 3 and executed on an Intel Core i5-3570K Processor.

The long runtime for the control unit when using elliptic curves with a large prime group order restricts the update rate in this scenario to frequencies between 1 and 10 Hz, while smaller prime group elliptic curves allow for up to 50 Hz update frequency. Again, we stress that smaller prime group curves do not provide security against a powerful malicious adversary, but can be employed to improve the integrity of computations susceptible to random hardware malfunctions. Typical update rates for flight control in large aircraft lie in the range of 25 Hz (Space Shuttle [48]) and 100 Hz (Boeing Fly-by-Wire system [13]).

We conclude that for simple control laws, reductions in prover runtime by ADSC-SNARK over previous approaches enable a usage in real-time systems with moderate timing requirements for protection against random hardware faults. For more complex control laws and with requirements to protect against intentional tampering, the prover runtimes currently still limit the applicability of this approach to systems with small update frequencies.

6 Related Work

This work is the first to explicitly consider proofs of stateful computations and on authenticated data, both at the same time without arithmetizing signatures or hashes. In this section, we discuss methods that achieve either one of these properties.

Starting with proofs on authenticated data, some works describe the folklore approach of arithmetizing an entire signature verification algorithm [24, 42] or parts thereof [60]. The first SNARK-based solution that does not require arithmetizing a signature verification algorithm is AD-SNARK by Backes et al. [3]. AD-SNARK comes in two flavors: A designated verifier version and a public verifier version. The latter is, unfortunately, not succinct, as proof size and verifier runtime are linear in the size of signed input data. More recently, SPHinx [33] was designed, which is a publicly verifiable SNARK for proofs on authenticated data with proof size constant in the size of signed input data. However, verification time is linear in the number of signed data items. VerITAS [25] aims to prove that only a set of allowed transformations was applied to a signed image. They present two modes, where the first mode is a very efficient hashing algorithm to be arithmetized. The second mode is similar to our construction, where the signer signs a commitment to the image. The prover then applies a modified SNARK proving algorithm, which is based on the Plonk proof system. The modified SNARK prover proves that the input-image corresponds to the commitment without a need to arithmetize the commitment algorithm.

A different line of works on proofs of computations over authenticated data are homomorphic message authentication codes [18, 19, 37] and their publicly verifiable counterpart: homomorphic signatures [14, 20]. However, these constructions are either concretely inefficient as they build on fully homomorphic encryption ([37]), or on cryptographic multi-linear maps ([19, 20]), or they are

not succinct ([14, 18]). So, either the proof size or the verification time increase at least linearly with the size of the computation. Furthermore, the class of supported computations is more restricted than SNARK-based solutions.

Next, there exist several works on proofs of stateful computations. Our notion of state-consistency stipulates schemes facilitating multiple proofs over a set of shared data which we refer to as the state. One line of work follows the folklore approach of arithmetizing collision-resistant hash functions [15] or more complex structures such as Merkle-trees [7, 9] or RSA acumulators [49]. These approaches significantly increase the size of the relation and therefore prover runtime.

Gepetto [24] introduces an approach that does not increase the size of the relation. It extends the Pinocchio SNARK [52] to a commit-and-prove scheme, enabling proofs on relations with a shared state. Campanelli et al. [17] extend this idea by making the commitment independent of the relation and the proof system, allowing proofs from different proving schemes to share state. There is a large line of work on various other commit-and-proof schemes that could be used for state consistency as well [32, 46, 59].

A somewhat orthogonal approach to the idea of stateful proofs is incrementally verifiable computation (IVC) [44, 50, 58] which also considers proofs over iterative computations with some shared state but requires that a single succinct proof guarantees the correctness of all iteration steps instead of a single step. While this is a stronger, useful property for many applications, practical implementations are concretely expensive, as they require either arithmetizing an entire proof verification algorithm, making use of proof aggregation schemes, or proving the folding of several SNARK relations, which adds additional overhead to the prover. Note that in our setting, we aim for the ability of the verifier to immediately check a computational output in each iteration.

7 Conclusion

In this paper, we have introduced ADSC-SNARKs, a generalization of SNARKs that add input data authentication and state consistency over multiple executions. Including these two properties in SNARKs is important for applications in the domain of cyber-physical control systems.

ADSC-SNARKs combine authentication of input data with state consistency in an efficient manner by careful modifications and optimizations to previous works on achieving the properties separately. Compared to a naïve solution, ADSC-SNARK achieve a 89× reduction in prover time. Compared to related, more efficient, approaches, ADSC-SNARK achieve better prover time, verification time and smaller proof size.

References

1. Ames, S., Hazay, C., Ishai, Y., Venkitasubramaniam, M.: Ligero: Lightweight Sublinear Arguments Without a Trusted Setup. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2087–2104, ACM (2017)

2. Andreica, T., Groza, B., Murvay, P.S.: Applications of Pairing-Based Cryptography on Automotive-Grade Microcontrollers. In: Gallina, B., Skavhaug, A., Schoitsch, E., Bitsch, F. (eds.) Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol. 11094, pp. 331–343, Springer International Publishing, Cham (2018)
3. Backes, M., Barbosa, M., Fiore, D., Reischuk, R.M.: ADSNARK: Nearly Practical and Privacy-Preserving Proofs on Authenticated Data. In: 2015 IEEE Symposium on Security and Privacy, pp. 271–286, IEEE (2015)
4. Bakken, B.H., Grande, O.S.: Automatic generation control in a deregulated power system. IEEE Transactions on Power Systems **13**(4), 1401–1406 (1998)
5. Barreto, P.S.L.M., Naehrig, M.: Pairing-Friendly Elliptic Curves of Prime Order. In: International workshop on selected areas in cryptography, pp. 319–331, Springer, Berlin, Heidelberg (2005)
6. Bellés-Muñoz, M., Isabel, M., Muñoz-Tapia, J.L., Rubio, A., Baylina, J.: Circom: A Circuit Description Language for Building Zero-Knowledge Applications. IEEE Transactions on Dependable and Secure Computing **20**(6), 4733–4751 (2023)
7. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E.: Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In: Proceedings of the 4th conference on Innovations in Theoretical Computer Science, pp. 401–413, ACM Press (2013)
8. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In: Proceedings of the 33rd Annual International Cryptology Conference, pp. 90–108, Springer, Berlin, Heidelberg (2013)
9. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Scalable Zero Knowledge via Cycles of Elliptic Curves. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014, Lecture Notes in Computer Science, vol. 8617, pp. 276–294, Springer Berlin Heidelberg (2014)
10. Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In: 23rd USENIX Security Symposium, pp. 781–796, USENIX Association (2014)
11. Bitansky, N., Canetti, R., Chiesa, A., Tromer, E.: From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, pp. 326–349, ACM Conferences, ACM (2012)
12. Bitansky, N., Chiesa, A., Ishai, Y., Paneth, O., Ostrovsky, R.: Succinct Non-interactive Arguments via Linear Interactive Proofs. In: Theory of Cryptography: 10th Theory of Cryptography Conference, pp. 315–333, Springer, Berlin, Heidelberg (2013)
13. Bleeg, R.: Commercial jet transport fly-by-wire architecture considerations. In: Digital Avionics Systems Conference, pp. 399–406, American Institute of Aeronautics and Astronautics (1988)
14. Boneh, D., Freeman, D.M.: Homomorphic Signatures for Polynomial Functions. In: Advances in Cryptogoly - EUROCRYPT 2011, pp. 149–168, Springer, Berlin, Heidelberg (2011)
15. Braun, B., Feldman, A.J., Ren, Z., Setty, S., Blumberg, A.J., Walfish, M.: Verifying computations with state. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 341–357, ACM (2013)

16. Bünz, B., Fisch, B., Szepieniec, A.: Transparent SNARKs from DARK Compilers. In: Canteaut, A., Ishai, Y. (eds.) Advances in Cryptology – EUROCRYPT 2020, pp. 677–706, Springer eBook Collection, Springer International Publishing and Imprint Springer (2020)
17. Campanelli, M., Fiore, D., Querol, A.: LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2075–2092, ACM (2019)
18. Catalano, D., Fiore, D.: Practical Homomorphic MACs for Arithmetic Circuits. In: Advances in Cryptology - EUROCRYPT 2013, pp. 336–352, Springer, Berlin, Heidelberg (2013)
19. Catalano, D., Fiore, D., Gennaro, R., Nizzardo, L.: Generalizing Homomorphic MACs for Arithmetic Circuits. In: Krawczyk, H. (ed.) Public-key cryptography, pp. 538–555, Lecture notes in computer science Security and cryptology, Springer (2014)
20. Catalano, D., Fiore, D., Warinschi, B.: Homomorphic Signatures with Efficient Verification for Polynomial Functions. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology - CRYPTO 2014, pp. 371–389, Lecture notes in computer science Security and cryptology, Springer (2014)
21. Chien, E., O' Murchu, L., Falliere, N.: W32.Duqu: The precursor to the next Stuxnet. In: 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats (2012)
22. Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In: Canteau, A., Ishai, Y. (eds.) Advances in Cryptology – EUROCRYPT 2020, pp. 738–768, Lecture Notes in Computer Science, Springer, Cham (2020)
23. Corps, S.G.: Airbus A320 Side Stick and Fly By Wire — An Update. In: SAE Technical Paper Series, pp. 1263–1275, SAE International (1986)
24. Costello, C., Fournet, C., Howell, J., Kohlweiss, M., Kreuter, B., Naehrig, M., Parno, B., Zahur, S.: Geppetto: Versatile Verifiable Computation. In: 2015 IEEE Symposium on Security and Privacy, pp. 253–270, IEEE (2015)
25. Datta, T., Chen, B., Boneh, D.: VerITAS: Verifying Image Transformations at Scale. Cryptology ePrint Archive (2024)
26. Driscoll, K., Hall, B., Paulitsch, M., Zumsteg, P., Sivencrona, H.: The real Byzantine Generals. In: The 23rd Digital Avionics Systems Conference, pp. 6.D.4–61–11, IEEE Operations Center (2004)
27. Eagen, L., Gabizon, A.: ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances. Cryptology ePrint Archive pp. 1–18 (2023)
28. Ericsson, G.N.: Cyber Security and Power System Communication—Essential Parts of a Smart Grid Infrastructure. IEEE Transactions on Power Delivery **25**(3), 1501–1507 (2010)
29. Falliere, N., O' Murchu, L., Chien, E.: W32.Stuxnet Dossier (2011)
30. Farineau, J., Letron, X.: Qualités de vol latéral d'un avion de transport civil équipé de commandes de vol électriques: expérience de l'Airbus A320. In: AGARD Conference Proceedings No.508, pp. 6.1–6.7 (1990)
31. Favre, C.: Fly-by-wire for commercial aircraft: the Airbus experience. International Journal of Control **59**(1), 139–157 (1994)
32. Fiore, D., Fournet, C., Ghosh, E., Kohlweiss, M., Ohrimenko, O., Parno, B.: Hash First, Argue Later. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1304–1316, ACM (2016)

33. Fiore, D., Tucker, I.: Efficient Zero-Knowledge Proofs on Signed Data with Applications to Verifiable Computation on Data Streams. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, ACM (2022)
34. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. Cryptology ePrint Archive pp. 1–34 (2019)
35. Galbraith, S.D., McKee, J.F., Valença, P.C.: Ordinary abelian varieties having small embedding degree. Finite Fields and Their Applications **13**(4), 800–814 (2007)
36. Gennaro, R., Gentry, C., Parno, B., Raykova, M.: Quadratic Span Programs and Succinct NIZKs without PCPs. In: Advances in Cryptology – EUROCRYPT 2013, pp. 626–645, Springer, Berlin, Heidelberg (2013)
37. Gennaro, R., Wichs, D.: Fully Homomorphic Message Authenticators. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology - ASIACRYPT 2013, pp. 301–320, Lecture Notes in Computer Science, Springer (2013)
38. Golovnev, A., Lee, J., Setty, S., Thaler, J., Wahby, R.S.: Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023, pp. 193–226, Lecture Notes in Computer Science, Springer Nature Switzerland and Imprint Springer (2023)
39. Grassi, L., Khovratovich, D., Rechberger, C., Roy, A., Schofnegger, M.: Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In: Proceedings of the 30th USENIX Security Symposium, pp. 519–535, USENIX Association (2021)
40. Groth, J.: On the Size of Pairing-Based Non-interactive Arguments. In: Advances in Cryptology – EUROCRYPT 2016, pp. 305–326, Springer Berlin Heidelberg (2016)
41. Jarmakiewicz, J., Parobczak, K., Maślanka, K.: Cybersecurity protection for power grid control infrastructures. International Journal of Critical Infrastructure Protection **18**, 20–33 (2017)
42. Kosba, A., Zhao, Z., Miller, A., Qian, Y., Chan, T.H.H., Papamanthou, C., Pass, R., Shelat, A., Shi, E.: C∅C∅: A Framework for Building Composable Zero-Knowledge Proofs. Cryptology ePrint Archive (2015)
43. Kothapalli, A., Setty, S.: HyperNova: Recursive Arguments for Customizable Constraint Systems. In: Reyzin, L., Stebila, D. (eds.) Advances in Cryptology – CRYPTO 2024, pp. 345–379, Lecture Notes in Computer Science, Springer Nature Switzerland and Imprint Springer (2024)
44. Kothapalli, A., Setty, S., Tzialla, I.: Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In: Advances in Cryptology - CRYPTO 2022, pp. 359–388, Springer, Cham (2022)
45. Lala, J.H., Harper, R.E.: Architectural Principles for Safety-Critical Real-Time Applications. Proceedings of the IEEE **82**(1), 25–40 (1994)
46. Lipmaa, H.: Prover-Efficient Commit-and-Prove Zero-Knowledge SNARKs. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) Progress in cryptology - AFRICACRYPT 2016, pp. 185–206, Lecture notes in computer science Security and cryptology, Springer (2016)
47. Maller, M., Bowe, S., Kohlweiss, M., Meiklejohn, S.: Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In: Cavallaro, L. (ed.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2111–2128, ACM Digital Library, Association for Computing Machinery (2019)

48. Minott, G.M., Peller, J.B., Cox, K.J.: Space Shuttle Digital Flight Control System. Advanced Control Technology and its Potential for Future Transport Aircraft pp. 271–294 (1976)
49. Ozdemir, A., Riad Wahby, Barry Whitehat, Dan Boneh: Scaling Verifiable Computation Using Efficient Set Accumulators. In: Proceedings of the 29th USENIX Security Symposium, pp. 2075–2092, USENIX Association (2020)
50. Paneth, O., Pass, R.: Incrementally Verifiable Computation via Rate-1 Batch Arguments. In: 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science, pp. 1045–1056, IEEE (2022)
51. Parno, B.: A Note on the Unsoundness of vnTinyRAM's SNARK. Cryptology ePrint Archive pp. 1–4 (2015)
52. Parno, B., Howell, J., Gentry, C., Raykova, M.: Pinocchio: Nearly Practical Verifiable Computation. In: 2013 IEEE Symposium on Security and Privacy, pp. 238–252, IEEE (2013)
53. Pedersen, T.P.: Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In: Feigenbaum, J. (ed.) Advances in Cryptology — CRYPTO '91, pp. 129–140, SpringerLink Bücher, Springer-Verlag Berlin Heidelberg (1992)
54. Reinhart, J., Luettig, B., Huber, N., Liedtke, J., Annighoefer, B.: Verifiable Computing in Avionics for Assuring Computer-Integrity without Replication. In: Digital Avionics Systems Conference 2023, pp. 1–10 (2023)
55. Setty, S.: Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020, Springer eBook Collection, vol. 12172, pp. 704–737, Springer International Publishing and Imprint Springer (2020)
56. Setty, S., Lee, J.: Quarks: Quadruple-efficient transparent zkSNARKs. Cryptology ePrint Archive (2020)
57. Traverse, P., Lacaze, I., Souyris, J.: Airbus Fly-By-Wire: A Total Approach To Dependability. In: Building the Information Society, pp. 191–212, Springer, Boston, MA (2004)
58. Valiant, P.: Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In: Theory of Cryptography: Fifth Theory of Cryptography Conference, pp. 1–18, Springer, Berlin, Heidelberg (2008)
59. Veeningen, M.: Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation. In: Joye, M., Nitaj, A. (eds.) Progress in Cryptology - AFRICACRYPT 2017, pp. 21–39, Lecture notes in computer science Security and cryptology, Springer (2017)
60. Wan, Z., Guan, Z., Zhou, Y., Ren, K.: zk-AuthFeed: How to Feed Authenticated Data into Smart Contract with Zero Knowledge. In: 2019 IEEE International Conference on Blockchain, pp. 83–90, IEEE (2019)
61. Yeh, Y.C.: Triple-triple redundant 777 primary flight computer. In: 1996 IEEE Aerospace Applications Conference. Proceedings, vol. 1, pp. 293–307 (1996)

A Security Proof

We prove that our construction satisfies the three properties completeness, knowledge-soundness, and succinctness following definitions 1, 2, and 3 in the *generic*

group model. Our proof follows the same type of blueprint and arguments as Groth [40].

Before diving into formal details, we start by sketching the intuition behind our proof. Note that we defer the treatment of zero-knowledge to Section B.

A.1 Proof Intuition

Verifier $\mathcal{V}$ checks the following relation between proof elements "in the exponent":

$$A_t \cdot B_t = \alpha \cdot \beta + \sum_{i \in \{0\} \cup \Phi} a_{t,i} \frac{P_i(z)}{\gamma} \cdot \gamma + C_t \cdot \delta + D_t \cdot \varepsilon + E_{t-1} \cdot \eta + E_t \cdot \kappa \quad (2)$$

Completeness is given by construction, which can be verified by inserting the terms for proof elements $A_t, B_t, C_t, ...$ in (2), and by comparing the result with QAP relation (1).

We show *knowledge-soundness* in two steps:
First, prover $\mathcal{P}$ is forced to build a proof from prover-key σ_p, as Equation 2 checks a relation depending on secret values $\alpha, \beta, z, ...$ The only information that $\mathcal{P}$ has on these secrets are the prover-key elements. As operations take place "in the exponent", $\mathcal{P}$ is restricted to only linear operations on prover-key elements σ_p, which are encoded as elements in $\mathbb{G}_1$ or $\mathbb{G}_2$. While it is possible to also evaluate the bilinear map on these elements, one can show that $\mathcal{P}$ does not gain any useful information from doing so (disclosure-freeness[40]).
Second, as $\mathcal{P}$ can only compute linear combinations of the prover-key, we can parameterize the set of all possible proofs it can produce. By inserting into (2) and comparing coefficients on both sides, one can show that all possible proofs satisfying (2) contain proof elements with coefficients a_i that also satisfy the original QAP. Hence, $\mathcal{P}$ must know these a_i that satisfy QAP. Similarly, (2) can only be satisfied, if those coefficients representing the state $(a_i)_{i \in S}$ in the current proof and coefficients representing the state update $(a_i)_{i \in S'}$ are equal to the coefficients in commitments c_{t-1} (proof element E_{t-1}) and c_t (E_t), which ensures state consistency. Finally, for (2) to hold, coefficients $(a_i)_{i \in X}$ must equal the committed coefficients in proof element D. By letting the verifier check the signature on proof element D, authenticity is ensured.

A.2 Preliminaries

We require additional security definitions for proving security of our ADSC-SNARKs. First, the construction of our authentication function (and therefore soundness) relies on digital signatures. Recall the standard security definition of a digital signature scheme with generator $(sk, pk) \leftarrow \mathsf{SigGen}(\lambda)$, signing algorithm $sig \leftarrow \mathsf{SigSign}(sk, m)$ and verification algorithm $b \leftarrow \mathsf{SigVerify}(pk, sig, m)$ for message m, private key sk, public key pk, and signature sig, such that:

Definition 4 (Correctness of Digital Signature Scheme). *For all security parameters λ and messages m:*

$$\Pr\left[\begin{array}{c}(sk, pk) \leftarrow \mathsf{SigGen}(\lambda) : \\ \mathsf{SigVerify}(pk, \mathsf{SigSign}(sk, m), m) = 1\end{array}\right] = 1.$$

Definition 5 (Security of Digital Signature Scheme). *A digital signature scheme* $(\mathsf{SigGen}, \mathsf{SigSign}, \mathsf{SigVerify})$ *is secure, if for all PPT adversaries $\mathcal{A}$ with oracle access to* $\mathsf{SigSign}$*:*

$$\Pr\left[\begin{array}{c}(sk, pk) \leftarrow \mathsf{SigGen}(\lambda); (m, sig) \leftarrow \mathcal{A}^{\mathsf{SigSign}(\cdot)}() : \\ \mathsf{SigVerify}(pk, sig, m) = 1 \wedge m \notin \tilde{M}\end{array}\right] \approx 0,$$

where $\tilde{M}$ is the set of queried messages by $\mathcal{A}$.

Second, we make use of a Pedersen vector commitment scheme for vector v of size n with binding properties:

Setup: $\sigma_c \leftarrow \mathsf{PedSetup}(\lambda)$:
Setup group $\mathbb{G}_1$ with size $|\mathbb{G}_1| = p$ and $|p| = \lambda$.
Pick: $T_i \leftarrow\!\!\$\ \mathbb{G}_1$ for $i \in [0, n]$, Set $\sigma_c = (T_i)_{i \in [0,n]}$

Commit: $(c, o) \leftarrow \mathsf{PedCommit}(\sigma_c, v)$:
Parse T_i from σ_c.
Pick: $o \leftarrow\!\!\$\ \mathbb{F}_p^*$, Compute $c = o \cdot T_0 + \sum_{i \in [1,n]} v_i \cdot T_i$.

Verify: $b \leftarrow \mathsf{PedVerify}(\sigma_c, c, v, o)$:
Parse T_i from σ_c.
Output $b = 1$ (accept), iff $c = o \cdot T_0 + \sum_{i \in [1,n]} v_i \cdot T_i$

A pedersen vector commitment is correct, iff:

Definition 6 (Correctness of Pedersen Vector Commitment). *For all security parameters λ and vectors v:*

$$\Pr\left[\begin{array}{c}\sigma_c \leftarrow \mathsf{PedSetup}(\lambda); \\ (c, o) \leftarrow \mathsf{PedCommit}(\sigma_c, v) : \\ \mathsf{PedVerify}(\sigma_c, c, v, o) = 1\end{array}\right] = 1.$$

A pedersen vector commitment is binding:

Definition 7 (Security of Pedersen Commitment). *For every PPT adversary $\mathcal{A}$:*

$$\Pr\left[\begin{array}{c}\sigma_c \leftarrow \mathsf{PedSetup}(\lambda); (c, v, o, \tilde{v}, \tilde{o}) \leftarrow \mathcal{A}(\sigma_c) : \\ \mathsf{PedVerify}(\sigma_c, c, v, o) = 1 \\ \wedge\, \mathsf{PedVerify}(\sigma_c, c, \tilde{v}, \tilde{o}) = 1 \\ \wedge v \neq \tilde{v}\end{array}\right] \approx 0.$$

A.3 Proof Details

We proceed by proving the properties of our ADSC-SNARK from section 3.

Completeness

Theorem 1. *Our* ADSC-*SNARK is complete.*

Proof. We show that for all $t \in [1, r]$, Verify will accept given outputs from Setup, Auth, and Prove for a satisfied relation R with state-consistency.
Step 1.1. Inserting A_t, B_t, C_t, D_t, E_t, and E_{t-1} in (2), multiplying out γ, δ, η, κ, and ε, and subtracting $\alpha\beta$ from both sides, we get

$$\begin{aligned}
&\sum_{i\in[0,m]} a_{t,i}u_i(z) \cdot \sum_{i\in[0,m]} a_{t,i}v_i(z) \\
&+ \beta \sum_{i\in[0,m]} a_{t,i}u_i(z) + \alpha \sum_{i\in[0,m]} a_{t,i}v_i(z) \\
&= \sum_{i\in[0,m]} a_{t,i}P_i(z) + h_t(z)t(z) - \sum_{i\in S} a_{t,i}\eta R_{s2s(i)} + \sum_{i\in S'} a_{t-1,i}R_i\eta.
\end{aligned}$$

Recall that function $s2s$ maps indices from states S to state updates S'. Condition $s_t = s'_{t-1}$ translates to $a_{t,i} = a_{t-1,s2s(i)}$ for $i \in S$. Therefore, the latter two sums are equal and cancel out. Writing out $P_i(z)$ as $\beta u_i(z) + \alpha v_i(z) + w_i(z)$, we get

$$\begin{aligned}
&\sum_{i\in[0,m]} a_{t,i}u_i(z) \cdot \sum_{i\in[0,m]} a_{t,i}v_i(z) \\
&+ \beta \sum_{i\in[0,m]} a_{t,i}u_i(z) + \alpha \sum_{i\in[0,m]} a_{t,i}v_i(z) \\
&= \sum_{i\in[0,m]} a_{t,i}\left(\beta u_i(z) + \alpha v_i(z) + w_i(z)\right) + h_t(z)t(z).
\end{aligned}$$

We further simplify to

$$\sum_{i\in[0,m]} a_{t,i}u_i(z) \cdot \sum_{i\in[0,m]} a_{t,i}v_i(z) - \sum_{i\in[0,m]} a_{t,i}w_i(z) = h_t(z)t(z).$$

This equation holds when the QAP is satisfied. Note that we do not need to consider a separate case for E_0, as E_0 in Setup is computed in the same way as E_t for $t > 0$ in Prove.
Step 1.2. According to the correctness property (Definition 4) of a signature scheme, a valid signature verifies:

$$\begin{aligned}
&\mathsf{SigVerify}(pk, sig_t, ([D_t]_1, t)) \\
&\qquad = \mathsf{SigVerify}(pk, \mathsf{SigSign}(sk, ([D_t]_1, t)), ([D_t]_1, t)) = 1.
\end{aligned}$$

Step 1.3. As both conditions which the verifier checks are satisfied in every case, it will always accept. □

Knowledge Soundness To prove knowledge-soundness, we assume the generic group model. That is, any adversary $\mathcal{A}$ is restricted to only performing group operations and applying the bilinear map on group elements.

Theorem 2. *In the generic group model, our* ADSC*-SNARK is knowledge-sound.*

Proof. First (step 2.1), following Groth we argue, that the reference-string (σ_p, σ_v) is disclosure-free and therefore $\mathcal{A}$ does not use the bilinear map. Following (steps 2.2 - 2.4), we show that a witness can be extracted, which satisfies state-consistency (step 2.5) and input data authenticity (steps 2.6 - 2.7).

Step 2.1. The reference-string in our construction is disclosure-free (see Definition 4 in [40]), the argument in the proof of Theorem 2 in [40]) applies: The reference-string elements can be regarded as multivariate polynomials in $\alpha, \beta, z, ...$, and any quadratic test on them (applying the bilinear map) evaluating to zero is either due to one of the following: 1.) The multivariate polynomials evaluate to zero for any $\alpha, \beta, z, ...$ This does not disclose any additional information, as the same test on any other reference-string (with different $\alpha, \beta, z, ...$) will also evaluate to zero. 2.) The multivariate polynomials evaluate to zero for a set of specific variables $\alpha, \beta, z, ...$ This case occurs with negligible probability due to the Schwartz-Zippel lemma.

Step 2.2. The verifier checks a relation between the proof elements involving secret values α, β, $\gamma, ...$ The only information about the secret values available to $\mathcal{A}$ are the elements of the reference-string (σ_p, σ_v) as well as $\{[T_i]_1\}_{i \in X}$ (as it has oracle access to Auth, allowing it to query arbitrary linear combinations of $[T_i]_1$). Due to the generic group model and the disclosure-freeness of the reference-string, we conclude, that $\mathcal{A}$ only outputs linear combinations of those elements as proof elements. To show that knowledge-soundness holds even if there were an efficient homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$, we assume that $\mathcal{A}$ has access to these elements in both $\mathbb{G}_1$ and $\mathbb{G}_2$. For example for the first element it knows $[\alpha]_1$ and $[\alpha]_2$. All possible linear combinations that $\mathcal{A}$ can output as proof elements can formally be considered as multi-variate Laurent polynomials where the randomly chosen secrets α, β, γ, δ, κ, η, ε, z, $\{T_i\}_{i \in X}$, $\{R_i\}_{i \in S'}$ are indeterminates. For simplicity, we will look at the corresponding exponents only, omitting the group element notation. We parameterize all possible polynomials with the elements of the reference-string and name coefficients according to the corresponding element. For example, the polynomial for C is:

$$C(\alpha, \beta, \gamma, \delta, \kappa, \eta, \varepsilon, z, \{T_i\}_{i \in X}, \{R_i\}_{i \in S'}) = C_\alpha \alpha + C_\beta \beta + \sum_{i \in [0,m]} C_{(u_i(z))} u_i(z) + \sum_{i \in [0,m]} C_{(v_i(z))} v_i(z) + \sum_{i \in S'} C_{(P_i(z)/\delta)} \frac{P_i(z)}{\delta} + \dots \ .$$

For A and B, we factor out A_α and B_β and mark the scaled coefficients with a dash ($A'_\beta = A_\beta/A_\alpha$, $B'_\alpha = B_\alpha/B_\beta$, ...) yielding:

$$A(\alpha, \beta, \ldots) = A_\alpha(\alpha + A'_\beta\beta + \sum_{i\in[0,m]} A'_{(u_i(z))}u_i(z) + \ldots).$$

We do the equivalent for B. We will show later that A_α and B_β cannot be zero.

We insert the polynomials in the verification equations. Consider the case that the resulting polynomial on the left side and the resulting polynomial on the right side of one of these equations do not have the same coefficients for their indeterminates. In this case, the probability that both polynomials evaluate to the same value when inserting the indeterminates is negligible due to the Schwartz-Zippel lemma. As a result, we can assume that coefficients on both sides of the equation are equal.

Step 2.3. We start with (2) for timestep t and look at proof elements A_t, B_t, C_t, D_t, E_t and E_{t-1}. To improve readability, we omit index t when it can be determined from the context.

Comparing coefficients of α^2, we get $A_\alpha B_\beta B'_\alpha = 0$, therefore one of the term's factors must be zero. Comparing coefficients of $\alpha\beta$, we get $A_\alpha B_\beta + A_\alpha A'_\beta B_\beta B'_\alpha = A_\alpha B_\beta(1 + A'_\beta B'_\alpha) = 1$. Therefore, neither A_α nor B_β can be zero. Thus, $B'_\alpha = 0$ and $A_\alpha B_\beta = 1$. Comparing coefficients of β^2, we get $A_\alpha A'_\beta B_\beta = A'_\beta = 0$. The polynomial from the left-hand side of the first verification equation then simplifies to:

$$LHS = \left(\alpha + \sum_{i\in[0,m]} A'_{(u_i(z))}u_i(z) + \sum_{i\in[0,m]} A'_{(v_i(z))}v_i(z) + \ldots\right) \times \left(\beta + \sum_{i\in[0,m]} B'_{(u_i(z))}u_i(z) + \sum_{i\in[0,m]} B'_{(v_i(z))}v_i(z) + \ldots\right).$$

Step 2.4. We now demonstrate that the remaining terms in A, which are not multiples of α, $u_i(z)$, $v_i(z)$, δ, ε, η or κ, must be zero, due to the left-hand side product resulting in terms with multiples of β that do not appear as indeterminates on the right-hand side: Comparing coefficients β/γ, we get $\sum_{i\in\{0\}\cup\Phi} A'_{(P_i(z)/\gamma)}P_i(z)/\gamma = 0$. Terms are zero similarly for coefficients β/δ, $\beta\gamma$, βT_i for any $i \in X$ and βR_i for any $i \in S'$. The same holds for the terms in B when comparing coefficients α/γ, α/δ, $\alpha\gamma$, αT_i for any $i \in X$ and αR_i for any $i \in S'$.

We now have

$$LHS = \begin{pmatrix}\alpha + \sum_{i\in[0,m]} A'_{(u_i(z))}u_i(z) + \sum_{i\in[0,m]} A'_{(v_i(z))}v_i(z)\\ +A'_\delta\delta + A'_\varepsilon\varepsilon + A'_\eta\eta + A'_\kappa\kappa\end{pmatrix} \times \begin{pmatrix}\beta + \sum_{i\in[0,m]} B'_{(u_i(z))}u_i(z) + \sum_{i\in[0,m]} B'_{(v_i(z))}v_i(z)\\ +B'_\delta\delta + B'_\varepsilon\varepsilon + B'_\eta\eta + B'_\kappa\kappa\end{pmatrix}.$$

Comparing the remaining terms with α on both sides, we get (recall that $P_i(z)$ contains terms $\alpha v_i(z)$):

$$\begin{aligned}
&\alpha \left(\sum_{i\in[0,m]} B'_{(u_i(z))} u_i(z) + \sum_{i\in[0,m]} B'_{(v_i(z))} v_i(z) \right) \\
&= \sum_{i\in\{0\}\cup\Phi} a_{t,i} \cdot \alpha v_i(z) + \sum_{i\in\Omega} C_{t,P_i(z)/\delta} \cdot \alpha v_i(z) \\
&+ \sum_{i\in S} C_{t,(P_i(z)-\eta R_{s2s(i)})/\delta} \cdot \alpha v_i(z) \\
&+ \sum_{i\in S'} C_{t,(P_i(z)-\kappa R_i)/\delta} \cdot \alpha v_i(z) \\
&+ \sum_{i\in X} C_{t,(P_i(z)-\varepsilon T_i)/\delta} \cdot \alpha v_i(z)
\end{aligned}$$

Renaming coefficients $C_{t,\dots}$ to $a_{t,i}$, we get:

$$\begin{aligned}
&\alpha \left(\sum_{i\in[0,m]} B'_{(u_i(z))} u_i(z) + \sum_{i\in[0,m]} B'_{(v_i(z))} v_i(z) \right) \\
&\qquad = \sum_{i\in[0,m]} a_i \alpha v_i(z).
\end{aligned}$$

We compare similarly the remaining terms with β on both sides. Then the left hand side simplifies to:

$$LHS = \begin{pmatrix} \alpha + \sum_{i\in[0,m]} a_i u_i(z) \\ +A'_\delta\delta + A'_\varepsilon\varepsilon + A'_\eta\eta + A'_\kappa\kappa \end{pmatrix} \times \begin{pmatrix} \beta + \sum_{i\in[0,m]} a_i v_i(z) \\ +B'_\delta\delta + B'_\varepsilon\varepsilon + B'_\eta\eta + B'_\kappa\kappa \end{pmatrix}.$$

Finally, comparing all terms with powers of z, we get:

$$\begin{aligned}
&\left(\sum_{i\in[0,m]} a_{t,i} u_i(z) \right) \cdot \left(\sum_{i\in[0,m]} a_{t,i} v_i(z) \right) \\
&\qquad = \sum_{i\in[0,m]} a_{t,i} w_i(z) + \sum_{i\in[0,d-2]} C_{t,(z^i t(z)/\delta)} z^i t(z)
\end{aligned}$$

Writing the last sum as $h(z)t(z)$, one can see that the $a_{t,i}$ for $i \in [0, m]$ satisfy the QAP, i.e. $(a_{t,i})_{i\in[1,m]} = (\phi_t, x_t, s_t, s'_t, \omega_t)$, $(\phi_t, x_t, s_t, s'_t, \omega_t) \in \mathrm{R}$. They can be extracted from $\mathcal{A}$, as the $a_{t,i}$ appear as coefficients of the proof elements.
Step 2.5. We now show that $\mathcal{V}$ accepting at iteration t implies $s_t = s'_{t-1}$.

Comparing all remaining terms with ηR_i for each $i \in S$, we get:

$$\forall i \in S : E_{t-1,R_{s2s(i)}} \eta R_{s2s(i)} - a_{t,i} \eta R_{s2s(i)} = 0$$
$$\Rightarrow \forall i \in S : E_{t-1,R_{s2s(i)}} = a_{t,i}.$$

Similarly, when comparing terms with κR_i for each $i \in S$, we get:

$$\begin{aligned}
&\forall i \in S : E_{t,R_{s2s(i)}} \kappa R_{s2s(i)} - a_{t,s2s(i)} \eta R_{s2s(i)} = 0 \\
&\Rightarrow \forall i \in S : E_{t,R_{s2s(i)}} = a_{t,s2s(i)} \\
&\Rightarrow \forall i \in S : E_{t-1,R_{s2s(i)}} = a_{t-1,s2s(i)}.
\end{aligned}$$

Combining both results gets us:

$$\forall i \in S : a_{t-1,s2s(i)} = a_{t,i} \Leftrightarrow s'_{t-1} = s_t.$$

Step 2.6. Next, we show that an accepting verifier $\mathcal{V}$ implies that the private input must have been authenticated. We first demonstrate, that coefficients in proof element D_t equal coefficients $a_{t,i}$ for $i \in X$.

Comparing all remaining terms with εT_i for each $i \in X$, we get:

$$\begin{aligned}
&\forall i \in X : D_{t,T_i} \varepsilon T_i - a_{t,i} \varepsilon T_i = 0 \\
&\Rightarrow \forall i \in X : D_{t,T_i} = a_{t,i}.
\end{aligned}$$

Step 2.7. As shown in step 2.4, if $\mathsf{Verify} = 1$, then $(\phi_t, x_t, s_t, s'_t, \omega_t) \in \mathrm{R}$, and as shown in step 2.5, if $\mathsf{Verify} = 1$ then $s_t = s'_{t-1}$. For knowledge-soundness it remains to be shown that if $\mathsf{Verify} = 1$, then the tuple (t, x_t) must be part of the set of queries to $\mathsf{Auth}(\cdot)$: $(t, x_t) \in \tilde{X}$. We do so by assuming that $\mathsf{Extract}$ can extract an x_t, such that $(t, x_t) \notin \tilde{X}$ and $\mathsf{Verify} = 1$. From that we construct another adversary $\mathcal{B}$ with oracle access to $\mathsf{SigSign}$, which either breaks security of the signature scheme or the binding property of the pedersen vector commitment scheme. We conclude that such an adversary does not exist.

We define the success probability of $\mathcal{B}$ as:

$$\Pr_{\mathcal{B}} = \Pr\left[\begin{array}{c}
(\sigma_a, \sigma_{aux}) \leftarrow \mathsf{Setup_a}(); \\
T_0 \leftarrow\!\!\$\ \mathbb{F}_p; T_i \leftarrow \mathsf{Parse}(\sigma_a); \\
\sigma_c = (T_0, \{T_i\}_{i \in X}); \\
(t, D, sig, x_t, \tilde{x}, o, \tilde{o}) \leftarrow \mathcal{B}^{\mathsf{SigSign}(\cdot)}(\sigma_c, \sigma_{aux}) : \\
\left(\begin{array}{c}
\mathsf{PedVerify}(\sigma_c, D, x_t, o) = 1 \\
\wedge\, \mathsf{PedVerify}(\sigma_c, D, \tilde{x}, \tilde{o}) = 1 \\
\wedge x_t \neq \tilde{x}
\end{array}\right) \\
\vee \left(\begin{array}{c}
\mathsf{SigVerify}(pk, sig, (D, t)) = 1 \\
\wedge (D, t) \notin \tilde{M}
\end{array}\right)
\end{array}\right],$$

where $\tilde{M}$ is the set of queries by $\mathcal{B}$ made to $\mathsf{SigSign}$.

$(t, D, sig, x_t, \tilde{x_t}, o, \tilde{o}) \leftarrow \mathcal{B}^{\mathsf{SigSign}(\cdot)}(\sigma_c, \sigma_{aux})$ works as follows:

Pick any $s'_0 \in \mathbb{F}_p^{|S|}$.
Compute

$$(\sigma_p, \sigma_v, c_0) \leftarrow \mathsf{Setup_{pv}}(s'_0, \sigma_{aux}).$$

Compute

$$((\pi_t, c_t, \phi_t); (x_t, s_t, s'_t, \omega_t)) \leftarrow (\mathcal{A}^{\mathsf{Auth}(\cdot)} \,||\, \mathsf{Extract})(\sigma_p, \sigma_v, t, c_0)$$

for any t, such that $\mathsf{Verify}(\sigma_v, \phi_t, \pi_t, c_t, c_{t-1}, t) = 1$ and $x_t \notin \tilde{X}$, where $\tilde{X}$ is the set of queries given to Auth.

Note, that $\mathcal{B}^{\mathsf{SigSign}(\cdot)}$ can answer the queries of $\mathcal{A}^{\mathsf{Auth}(\cdot)}$ to Auth by using the information from σ_{aux} and the oracle access to $\mathsf{SigSign}$. Let $\tilde{M}$ be the set of queries made by $\mathcal{B}$ to the signing oracle.

Now, $\mathcal{B}$ distinguishes two cases:
a.) If there exists $\tilde{x} \in \tilde{X}$, such that $x_t \neq \tilde{x}$ and $\sum_{i \in X} x_{t,i} T_i = \sum_{i \in X} \tilde{x}_i T_i$, output:

$$t, D = \sum_{i \in X} x_{t,i} T_i, \quad o = \tilde{o} = 0, \quad sig = \{\}.$$

b.) Otherwise, parse sig from π_t and output

$$t, D = \sum_{i \in X} x_{t,i} T_i, \quad sig, \quad x_t = \tilde{x_t} = o = \tilde{o} = \{\}.$$

We see, that if $\mathcal{A}$ is successful, either of the cases a.) or b.) occurs, resulting in either

$$\mathsf{PedVerify}(\sigma_c, D, x_t, o) = 1 \wedge \mathsf{PedVerify}(\sigma_c, D, \tilde{x}, \tilde{o}) = 1 \wedge x_t \neq \tilde{x_t}$$

to hold (case a), or

$$\mathsf{SigVerify}() = 1$$

to hold (case b).
Therefore, by construction, $\Pr_{\mathcal{B}} \approx \Pr_{\mathcal{A}}$.

It is now easy to see, that we can derive from $\mathcal{B}$ another adversary $\mathcal{C}$, which tries to break the binding property of a pedersen commitment of Definition 7. It's success probability is $\Pr_{\mathcal{C}}$.
$(c, v, o, \tilde{v}, \tilde{o}) \leftarrow \mathcal{C}(\sigma_c)$ works as follows:
Parse $\{T_i\}$ from σ_c
Compute σ_{aux} as in $\mathsf{Setup_a}$, but use T_i from σ_c, instead of sampling them.
Compute $(t, D, sig, x_t, \tilde{x_t}, o, \tilde{o}) \leftarrow \mathcal{B}^{\mathsf{SigSign}(\cdot)}(\sigma_c, \sigma_{aux})$.
Set $c = D, v = x_t, \tilde{v} = \tilde{x_t}$.
Similarly, we can derive from $\mathcal{B}$ an adversary $\mathcal{D}$ with oracle access to $\mathsf{SigSign}$, which tries to break the security of the signature scheme of Definition 5. It's success probability is $\Pr_{advd}$.
$(m, sig) \leftarrow \mathcal{D}^{\mathsf{SigSign}(\cdot)}()$ works as follows: Compute

$$(\sigma_a, \sigma_{aux}) \leftarrow \mathsf{Setup_a}()$$
$$T_0 \leftarrow\!\!\$\, \mathbb{F}_p, T_i \leftarrow \mathsf{Parse}(\sigma_a)$$
$$\sigma_c = (T_0, \{T_i\}_{i \in X}).$$

Run $\mathcal{B}$

$$(t, D, sig, x_t, \tilde{x_t}, o, \tilde{o}, t) \leftarrow \mathcal{B}^{\mathsf{SigSign}(\cdot)}(\sigma_c, \sigma_{aux}).$$

Set $m = (D, t)$.
Output m, sig.

As we assume that the pedersen vector commitment is binding (Definition 7), $\mathcal{C}$ has negligible chance of success: $\Pr_{\mathcal{C}} \approx 0$. As we also assume, that the signature scheme is secure (Definition 5), $\mathcal{D}$ has also negligible chance of success: $\Pr_{\mathcal{D}} \approx 0$. Both $\mathcal{C}$ and $\mathcal{D}$ having negligible chance of success means that also $\Pr_{\mathcal{B}} \approx 0$. It follows that $\Pr_{\mathcal{A}} \approx \Pr_{\mathcal{B}} \approx 0$. □

Theorem 3. *Our* ADSC*-SNARK is succinct.*

Proof. A proof π consists of a fixed number of 6 group elements and a signature. The algorithm Verify does $1 + |\phi|$ scalar multiplications, 6 pairing operations, 5 field multiplications, a signature verification and 2 comparisons. □

B Zero-Knowledge

We present a variation of our main ADSC-SNARK construction which is also zero-knowledge.

B.1 Zero-Knowledge Definition

We let algorithm Setup output a trapdoor τ which is passed to a simulator Sim. The simulator does not receive any of the private input x, states s and s' or witness ω. If adversary $\mathcal{A}$ will not be able to distinguish simulated from real proofs, no information about x, s, s' or ω is leaked by a real proof.

Definition 8 (Zero-knowledge of ADSC-SNARK). *For all* $\lambda \in \mathbb{N}$, $\mathrm{pp} \leftarrow \mathsf{Gen}(\lambda)$*, there exists a PPT algorithm* $(\pi_t, c_t) \leftarrow \mathsf{Sim}(\tau, \phi_t, c_{t-1}, t)$*, such that for all* $(a_1, ..., a_r) \in \mathrm{R}^r$*, with* $a_t = (\phi_t, x_t, s_t, s'_t, \omega_t)$*, and* $s'_0 \in \mathbb{H}^{|S|}$ *such that* $\bigwedge_{t=1}^{r}(s_t = s'_{t-1})$*, and for all adversaries* $v \leftarrow \mathcal{A}(\sigma_p, \sigma_v, \sigma_a, \tau, \Pi, C)$*:*

$$\Pr\left[\begin{array}{c}(\sigma_p, \sigma_v, \sigma_a, c_0, \tau) \leftarrow \mathsf{Setup}(s'_0);\\ \nu_t \leftarrow \mathsf{Auth}(\sigma_a, t, (a_{t,i})_{i \in X}) \,\mathrm{for}\, t \in [1, r]\,;\\ (\pi_t, c_t, p_t) \leftarrow \mathsf{Prove}(\sigma_p, a_t, \nu_t, p_{t-1})\\ \mathrm{for}\, t \in [1, r]:\\ \mathcal{A}(\sigma_p, \sigma_v, \sigma_a, \tau, (\pi_1, \pi_2, ..., \pi_t), (c_0, c_1, ..., c_t)) = 1\end{array}\right]$$

$$= \Pr\left[\begin{array}{c}(\sigma_p, \sigma_v, \sigma_a, c_0, \tau) \leftarrow \mathsf{Setup}(s'_0);\\ (\pi_t, c_t) \leftarrow \mathsf{Sim}(\tau, \phi_t, c_{t-1}, t) \,\mathrm{for}\, t \in [1, r]:\\ \mathcal{A}(\sigma_p, \sigma_v, \sigma_a, \tau, (\pi_1, \pi_2, ..., \pi_t), (c_0, c_1, ..., c_t)) = 1\end{array}\right].$$

B.2 Zero-Knowledge Construction

To make proofs zero-knowledge, the authenticating party and the prover add randomization masks b_A, b_B, b_C, b_D, b_E to the proof. These make the group elements in the proof indistinguishable from randomly drawn group elements. Additionally, the randomization masks cancel out in the verification equation, preserving completeness of the SNARK. For the randomization masks to cancel out, the prover needs to know $b_{E,t-1}$ from the previous iteration, therefore it stores b_E between iterations.

The zero-knowledge ADSC-SNARK construction is shown in Figure 6. Differences to the version without the zero-knowledge property are highlighted.

B.3 Security Proof

We show that the zero-knowledge ADSC-SNARK is complete, knowledge-sound, succinct, and zero-knowledge following definitions 1, 2, 3, and 8.

Theorem 4. *The zero-knowledge* ADSC-*SNARK is complete.*

Proof. When comparing the zero-knowledge variant to the main variant, it is easy to see, that the additional randomization masks b_A, b_B, b_C, b_D, b_E, that are added to the proof elements by Auth and Prove cancel out in the verification equations. □

Theorem 5. *The zero-knowledge* ADSC-*SNARK is zero-knowledge.*

Proof. We provide a simulator and show that both real and simulated proofs satisfy the verification equations and demonstrate that real and simulated proofs are distributed equally.

Simulate: $(\pi_t, c_t) \leftarrow \mathsf{Sim}(\tau, \phi_t, c_{t-1}, t)$
Set $E_{t-1} = c_{t-1}$.
Pick $A_t, B_t, D_t, E_t, \leftarrow\!\!\$\ \mathbb{F}_p$.
Compute $C_t =$
$\frac{1}{\delta}\left(A_t B_t - \alpha\beta - \sum_{i\in\{0\}\cup\Phi} a_{t,i} P_i(z) - D_t\varepsilon - E_{t-1}\eta - E_t\kappa\right)$.
Compute $sig = \mathsf{SigSign}(sk, ([D_t]_1, t))$.
Set $\pi_t = ([A_t]_1, [B_t]_2, [C_t]_1, [D_t]_1, sig)$, $c_t = [E_t]_1$.

Step 5.1. Real proofs verify due to completeness (Theorem 4). From construction, it is straight forward to see that simulated proofs verify.
Step 5.2. The elements A_t, B_t, D_t, E_t for $t \in [1, r]$ of both real and simulated proofs are uniformly and randomly distributed. The remaining proof element C_t is uniquely determined by the first verification equation and *sig* was derived from (D_t, t) using SigSign in both the real and simulated proofs. □

Theorem 6. *In the generic group model, our zero-knowledge* ADSC-*SNARK is knowledge-sound.*

$(\sigma_p,\sigma_v,\sigma_a,c_0,\tau) \leftarrow \mathsf{Setup}(s'_0)$

$T_i \leftarrow\!\$\, \mathbb{F}_p^* \text{for} i \in X, \quad R_i \leftarrow\!\$\, \mathbb{F}_p^* \text{for} i \in S', \quad \alpha,\beta,\gamma,\delta,\kappa,\eta,\varepsilon,z \leftarrow\!\$\, \mathbb{F}_p^*,$

$(sk,pk) \leftarrow \mathsf{SigGen}()$

$$\sigma_{p,1} = \left(\begin{array}{c} \alpha,\beta,\delta,\{u_i(z)\}_{i\in[0,m]},\{v_i(z)\}_{i\in[0,m]},\left\{\frac{P_i(z)}{\delta}\right\}_{i\in\Omega},\left\{\frac{P_i(z)-\eta R_{s2s(i)}}{\delta}\right\}_{i\in S}, \\ \left\{\frac{P_i(z)-\kappa R_i}{\delta}\right\}_{i\in S'},\left\{\frac{P_i(z)-\varepsilon T_i}{\delta}\right\}_{i\in X},\left\{\frac{z^i t(z)}{\delta}\right\}_{i\in[0,d-2]},\{R_i\}_{i\in S'},\varepsilon,\eta,\kappa \end{array} \right)$$

$$\sigma_{p,2} = \left(\beta,\delta,\{v_i(z)\}_{i\in[0,m]}\right), \quad \sigma_{v,1} = \left(\left\{\frac{P_i(z)}{\gamma}\right\}_{i\in\{0\}\cup\Phi}\right), \quad \sigma_{v,2} = (\gamma,\delta,\varepsilon,\eta,\kappa).$$

$$\sigma_p = \left([\sigma_{p,1}]_1,[\sigma_{p,2}]_2\right), \quad \sigma_v = \left([\sigma_{v,1}]_1,[\sigma_{v,2}]_2,[\alpha\beta]_T,pk\right),$$

$$\sigma_a = \left(sk,\{[T_i]_1\}_{i\in X},[\delta]_1\right), \quad c_0 = [E_0]_1 = \sum_{i\in S'} s'_{0,\mathrm{pos}_{S'}(i)}[R_i]_1,$$

$$\tau = (\alpha,\beta,\gamma,\delta,\kappa,\eta,\varepsilon,z,\{R_i\}_{i\in S'},sk).$$

$\nu_t \leftarrow \mathsf{Auth}(\sigma_a,t,x_t)$

$$b_D \leftarrow\!\$\, \mathbb{F}_p, D = \sum_{i\in X} a_i T_i + b_D\delta, sig = \mathsf{SigSign}(sk,([D]_1,t))\nu_t = \left([D]_1,b_D,sig\right)$$

$(\pi_t,c_t,p_t) \leftarrow \mathsf{Prove}(\sigma_p,a_t,\nu_t,p_{t-1})$

$$b_A,b_B,b_{t,E} \leftarrow\!\$\, \mathbb{F}_p, \quad b_{t-1,E} = \begin{cases} 0, \text{if } p_{t-1} = \{\} \\ p_{t-1}, \text{else} \end{cases}, \quad p_t = b_{t,E}$$

$$h(Z) = \left(\sum_{i\in[0,m]} a_{t,i}u_i(Z) \cdot \sum_{i\in[0,m]} a_{t,i}v_i(Z) - \sum_{i\in[0,m]} a_{t,i}w_i(Z) \right) / t(Z).$$

$$A = \alpha + \sum_{i\in[0,m]} a_i u_i(z) + b_A\delta, \quad B = \beta + \sum_{i\in[0,m]} a_i v_i(z) + b_B\delta,$$

$$C = \sum_{i\in\Omega} \frac{a_i P_i(z)}{\delta} + \sum_{i\in S} \frac{a_i\left(P_i(z)-\eta R_{s2s(i)}\right)}{\delta} + \sum_{i\in S'} \frac{a_i(P_i(z)-\kappa R_i)}{\delta}$$
$$+ \sum_{i\in X} \frac{a_i(P_i(z)-\varepsilon T_i)}{\delta} + \frac{h(z)t(z)}{\delta}$$
$$+ b_B A + b_A B - b_A b_B \delta - b_D\varepsilon - b_{t-1,E}\eta - b_{t,E}\kappa, \quad E = \sum_{i\in S'} a_i R_i + b_{t,E}\delta,$$

$$\pi_t = ([A]_1,[B]_2,[C]_1,[D]_1,sig), \quad c_t = [E]_1$$

$v \leftarrow \mathsf{Verify}(\sigma_v,\phi_t,\pi,c_t,c_{t-1},t)$

Accept, iff $[A_t]_1 \cdot [B_t]_2 = [\alpha\beta]_T + \left(\sum_{i\in\{0\}\cup\Phi} a_{t,i} \left[\frac{P_i(z)}{\gamma}\right]_1 \right) \cdot [\gamma]_2 + [C_t]_1 \cdot [\delta]_2$
$$+ [D_t]_1 \cdot [\varepsilon]_2 + [E_{t-1}]_1 \cdot [\eta]_2 + [E_t]_1 \cdot [\kappa]_2$$
and $\mathsf{SigVerify}(pk,sig_t,([D_t]_1,t)) = 1.$

Fig. 6. Zero-Knowledge ADSC-SNARK construction.

Proof. The verifier algorithm of the zero-knowledge ADSC-SNARK is the same as the verifier algorithm of the main ADSC-SNARK. The prover-key of the zero-knowledge ADSC-SNARK has additional elements $([\beta]_1, \left[\{v_i(z)\}_{i\in[0,m]}\right]_1, \ldots)$. However, these additional elements were already part of the prover-key or verifier-key of the main ADSC-SNARK version, but represented in a different group (e.g. $[\beta]_2$ was part of the prover-key of the main construction). As the proof for Theorem 2 is valid, even if there existed an efficiently computable homomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$, it follows from Theorem 2, that also the zero-knowledge ADSC-SNARK is sound. □

Theorem 7. *The zero-knowledge* ADSC-*SNARK is succinct.*

Proof. The structure of the proof and the verifier algorithms of the zero-knowledge ADSC-SNARK and the main ADSC-SNARK are the same. From Theorem 3 it follows that also the zero-knowledge variant is succinct. □

C Elliptic curve parameters

We list the parameters of the elliptic curves that were used for benchmarking ADSC-SNARK and the presented example use case.

C.1 Barreto-Naehring elliptic curves

Barreto-Naehring elliptic curves are described in [5]. The curves used in our benchmarks and examples are defined over base field prime q. They have prime group order r and embedding degree k. The prime group order r is of form $2^s \cdot r' + 1$, where r' is uneven (s is referred to as the two-adicity of prime field $\mathbb{F}_r$).

Groups $\mathbb{G}_1$ are the elliptic curve points defined over prime field $\mathbb{F}_q$ for the curve equation

$$y^2 = x^3 + b.$$

Groups $\mathbb{G}_2$ are defined over extension field $\mathbb{F}_{q^2} = \mathbb{F}_q[u]/(u^2 - \beta)$ for curve equation

$$y^2 = x^3 + b/\xi.$$

The parameters are listed in Table 5.

C.2 Galbraith-McKee-Valença elliptic curves

Galbraith-McKee-Valença elliptic curves are described in [35]. The curves used in our benchmarks and examples are defined over base field prime q. They have prime group order r and embedding degree k. The prime group order r is of form $2^s \cdot r' + 1$, where r' is uneven (s is referred to as the two-adicity of prime field $\mathbb{F}_r$).

Table 5. Parameters of used Barreto-Naehring elliptic curves

	BN254 [10]
k	12
r	21888242871839275222246405745257275088548364400416034343 698204186575808495617
q	21888242871839275222246405745257275088696311157297823662 689037894645226208583
β	-1
b	3
ξ	$9+u$
s	28
	BN183
k	12
r	68047597488463554058305827860110329707849460752664494 09
q	68047597488463554058305827886196264133984226022552364 23
β	-1
b	3
ξ	$2+u$
s	30
	BN124
k	12
r	17000133324792832058895897937997463553
q	17000133324792832063019019729102503239
β	-1
b	3
ξ	$5+u$
s	25

Groups $\mathbb{G}_1$ are the elliptic curve points defined over prime field $\mathbb{F}_q$ for the curve equation in twisted edwards form

$$a \cdot x^2 + y^2 = 1 + d \cdot x^2 y^2.$$

Groups $\mathbb{G}_2$ are defined over extension field $\mathbb{F}_{q^3} = \mathbb{F}_q[u]/(u^3 - \beta)$ for curve equation

$$a\xi \cdot x^2 + y^2 = 1 + d\xi \cdot x^2 y^2.$$

The parameters are listed in Table 6.

Table 6. Parameters of used Galbraith-McKee-Valença elliptic curves

	GMV181 [10]
k	6
r	155251103010243025123680156134462199326192089757122560 1
q	621004412040972100494720624088597827452375126979379200 1
β	61
a	1
d	600581931845324488256649384912508268813600056237543024
ξ	u
s	31
	GMV97
k	6
r	141455844224742490147094691841
q	565823376898968604518330826753
β	5
a	5
d	4829968250478157739833804867 79
ξ	u
s	15
	GMV58
k	6
r	211006452744585217
q	844025809322115073
β	10
a	5
d	579073710274753001
ξ	u
s	19