

# SNARKs for Stateful Computations on Authenticated Data

Johannes Reinhart<sup>1</sup> Erik-Oliver Blass<sup>2</sup> Bjoern Annighoefer<sup>1</sup>

<sup>1</sup> University of Stuttgart, Stuttgart, Germany

{johannes.reinhardt, bjoern.annighoefer}@ils.uni-stuttgart.de

<sup>2</sup> Airbus, Munich, Germany

erik-oliver.blass@airbus.com

**Abstract.** We present a new generalization of (zk-)SNARKs combining two additional features at the same time. Besides the verification of correct computation, our new SNARKs 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 SNARKs are specifically suited to applications in cyber-physical control systems, where computations are periodically carried out and need to be checked immediately. 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. Additionally, we present new optimizations to reduce proof size, prover time, and verification time in our setting. With our construction, prover runtime improves significantly over the baseline by a factor of 89. Verification time is 70% less for computations on authenticated data and 33% less for stateful computations. To demonstrate relevance and practicality, we implement and benchmark our new SNARKs in a sample real-world scenario with a (simple) quadcopter flight control system.

## 1 Introduction

In many applications, one party must demonstrate to others that it has 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  $\pi$ . The idea is that any verifier  $\mathcal{V}$  can use  $\pi$  and check whether  $y$  is the correct output with respect to  $x$  and  $f$ . Two key properties towards practical efficiency are that verification of  $\pi$  should be more efficient than re-computing  $y$ , and that the size of  $\pi$  is small, i.e., smaller than the size of input  $x$ . 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) [1–15]. 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 recent (zk-)SNARKs, combining two additional properties.

*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. 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* In many real-world scenarios, applications are stateful, and computations  $f$  depend 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 compute arbitrary outputs  $y_t$  and present them to  $\mathcal{V}$ . Thus, the second property we target in this paper is for  $\mathcal{P}$  to create a proof  $\pi_t$  allowing  $\mathcal{V}$  to efficiently verify the correctness of  $y_t$  and  $s_t$  in each 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  $\pi_t$  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.

We dub our construction ADSC-SNARKs, generalized SNARKs on Authenticated Data with State Consistency. An immediate application for ADSC-SNARKs are digital controllers in a cyber-physical system where input  $x_t$  is a tuple of measurements of some physical quantities, and  $y_t$  is a tuple of commands to actuators that manipulate the physical system to be controlled. In most such cases, the controller must periodically process sensor input, update its internal state and provide output commands. 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 immediately in each iteration.

*Related Work* While we discuss related work in detail later in Section 6, we briefly summarize the differences of our setting. Here, the (computationally limited) verifying party  $\mathcal{V}$  checks the output  $y_t$  of an iterative stateful computation in every single step  $t$ , immediately when  $y_t$  is available. Ideally,  $\mathcal{V}$  is not required to see  $s_t$  or  $x_t$ , as they can be large or private. SNARKs for incrementally verifiable computation based on recursive proofs [16] or folding schemes [17–19] aim for a verifier to check the entire iterative computation once at the final step within a single proof. This is a stronger requirement compared to our setting. They achieve this by additionally proving in each iteration the correctness of a proof verification of the previous iteration or the correctness of compressing (folding) two computational steps in each iteration. Constructions for composing SNARKs [20, 21] use commit-and-prove SNARKs to prove the conjunction, disjunction or functional composition of two or more computations  $f_1, f_2, \dots$ . These fit our setting better, because they can be used to prove stateful computations without recursion or folding. Our construction is therefore based on the latter approach, but specifically tailored to the setting of iterative stateful computations.

Previous SNARKs for computations on authenticated data [22–24] do not consider stateful computations. 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, adding the state consistency property is non-trivial. For example the designated-verifier SNARK in [22] uses one-time message authentication codes for the input data, which would be insecure in an iterative setting without additional measures.

Several works present proofs about authenticated data by arithmetizing and proving a signature verification algorithm [25–27]. Similarly, several works present proofs about computations with state by arithmetizing and proving hash functions [27–30]. In practice, these approaches lead to long proving times. In this paper, we call approaches that arithmetize hashes or verification algorithms as *folklore* approaches.

Tables 1 and 2 compare the performance and properties of our ADSC-SNARKs with related work.

*This paper* We present a new scheme dubbed *Authenticated Data and State Consistent Succinct Non-Interactive Arguments of Knowledge* (ADSC-SNARKs) which allows to efficiently verify the correctness of each value  $y_t$  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  $89\times$  for a circuit with  $2^{14}$  states and inputs. The C++ implementation is open source and freely available<sup>3</sup>.

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.

<sup>3</sup> <https://github.com/johannes-reinhart/adsc-snark>

**Table 1.** Comparison of our ADSC-SNARK with related approaches, part 1.  $|X|$ : number of private inputs,  $|S|$ : number of states,  $|\Omega|$ : size of witness,  $\text{MSM}_{\mathbb{G}_1}(n)$ : multiscalar multiplication of  $n$   $\mathbb{G}_1$  elements Overhead means difference in runtime or size of new SNARK compared to its Base SNARK.

Approach	Base SNARK	Arithmetization <sup>1</sup> Overhead	Prover Overhead
ADSC-SNARK (ours)	Groth16 [7]	—	$1 \times \text{MSM}_{\mathbb{G}_1}( S )$
Folklore ("Strawman")	Groth16 [7]	2126 $+72 X $ $+144 S $	—
AD-SNARK [22] desig. verifier	BCTV14 [6]	$ X $	$1 \times \text{MSM}_{\mathbb{G}_1}( X )$
AD-SNARK [22] pub. verifier	BCTV14 [6]	$ X $	$1 \times \text{MSM}_{\mathbb{G}_1}( X )$
SPHinx [23]	Marlin [13]	—	$1 \times \text{MSM}_{\mathbb{G}_1}( \Omega )$ $+4 \times \text{MSM}_{\mathbb{G}_1}( X )$ $+1 \times \text{MSM}_{\mathbb{G}_1}( X  +  \Omega )$
Geppetto [20] SC-SNARK	Pinocchio [4]	$2 \times  S $	$3 \times \text{MSM}_{\mathbb{G}_1}( S )$
LegoGro [21] SC-SNARK	Groth16 [7]	$2 \times  S $	$1 \times \text{MSM}_{\mathbb{G}_1}( S )$ $+1 \times \text{MSM}_{\mathbb{G}_1}(2 S )$

<sup>1</sup> Arithmetization overhead increases prover runtime.

**Table 2.** Comparison of our ADSC-SNARK with related approaches, part 2.  $|X|$ : number of private inputs,  $\mathbb{F}_p$ : prime field element,  $\mathbb{G}_1$ : curve group 1 element,  $\mathbb{G}_2$ : curve group 2 element,  $sig$ : signature,  $MSM_{\mathbb{G}_1}(n)$ : multiscalar multiplication of  $n$   $\mathbb{G}_1$  elements, P: pairing evaluation, SV: signature verification. Overhead means difference in runtime or size of new SNARK compared to its Base SNARK.

Approach	Verifier Overhead	Proof Overhead	Public Auth. Verif.	State Data	Consist.
ADSC-SNARK (ours)	$3 \times P$ $+1 \times SV$	$2 \times \mathbb{G}_1$ $+1 \times sig$	✓	✓	✓
Folklore ("Strawman")	—	$1 \times \mathbb{F}_p$	✓	✓	✓
AD-SNARK [22] desig. verifier	$1 \times MSM_{\mathbb{G}_1}( X )$ $2 \times P$	$3 \times \mathbb{G}_1$	—	✓	—
AD-SNARK [22] pub. verifier	$1 \times MSM_{\mathbb{G}_1}( X )$ $+( X  + 4) \times P$ $+ X  \times SV$	$3 \times \mathbb{G}_1$ $+ X  \times \mathbb{G}_2$ $+ X  \times sig$	✓	✓	—
SPHinx [23]	$7 \times P$ $+2 \times MSM_{\mathbb{G}_1}( X )$	$15 \times \mathbb{G}_1$ $10 \times \mathbb{F}_p$	✓	✓	—
Geppetto [20] SC-SNARK	$1 \times P$	$3 \times \mathbb{G}_1$	✓	—	✓
LegoGro [21] SC-SNARK	$4 \times P$	$3 \times \mathbb{G}_1$	✓	—	✓

- 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 extensive evaluation that benchmarks ADSC-SNARK and compares its performance to `LegoGro16` by Campanelli et al. [21], `AD – SNARK` by Backes et al. [22] 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 and cyber-physical 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.

*Zero-Knowledge* As with related work [4, 6, 7], also our ADSC-SNARK construction proves in zero-knowledge by adding randomization masks to the proof. To ease the exposition, we will focus on describing ADSC-SNARKs *without* the zero-knowledge property, and we prove only completeness and soundness. 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  $\pi_t$  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 [27–30].

To overcome this drawback, we follow the observation by Campanelli et al. [21] that the SNARK by Groth [7] 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 commitment can be linked to a commit-carrying SNARK, similarly to our techniques for state-consistency described above. In addition to verifying the proof  $\pi$ ,  $\mathcal{V}$  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. [24] to make proofs on signed images. However, they base their construction on the Plonk [10] proof system, which results in a larger proof size compared to our construction.

## 1.2 Notation

In this paper,  $\lambda$  denotes the computational security parameter. For instantiating our scheme, we use prime-order groups  $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, |\mathbb{G}_1|, |\mathbb{G}_2|, |\mathbb{G}_T| \in \text{poly}(\lambda)$  together with a Type-3 bilinear map  $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ . We denote a group element  $\alpha$  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  $\alpha$  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.

For a randomized algorithm  $F(x; r_{\S})$  with input  $x$  and random coins  $r_{\S}$ ,  $y \leftarrow F(x)$  denotes random sampling of  $r_{\S}$  and then assigning the output of  $F(x; r_{\S})$  to  $y$ . We denote the assignment of multiple elements using independent random coins  $r_{\S}$  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  $r_{\S}$  and then assigning the result of applying  $F$  to  $x$  with  $r_{\S}$  to  $y$ , and assigning the result of applying  $H$  to the same input  $x$  and the same random coin  $r_{\S}$  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  $y \leftarrow_{\S} S$ .

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

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

For a field  $\mathbb{F}$ , we write  $\mathbb{F}^*$  for the set of invertible elements of  $\mathbb{F}$ . A vector or tuple with a lower index  $i$  denotes the  $i^{\text{th}}$  element of that vector or tuple. Similarly, we distinguish variables for different time steps using index  $t$ .

We write  $[a, b]$  for the set  $\{x | a \leq x \leq b\}$ . For some set  $\{a_{i_1}, \dots, a_{i_n}\}$ , let  $\{i_1, \dots, i_n\}$  be the corresponding index set. For an index set  $S$ , we write  $\text{pos}_S(i)$  to order indices  $i$  by size, i.e., for  $S = \{3, 4\}$ ,  $\text{pos}_S(3) = 1$ ,  $\text{pos}_S(4) = 2$ .

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  $R_\lambda$  and security parameter  $\lambda$ .

### 2.1 Setup

We assume that  $R_\lambda$  can be generated by a generator  $\text{pp} \leftarrow \text{Gen}(\lambda)$  outputting public-parameters  $\text{pp}$ , such that  $\text{pp}$  include a field  $\mathbb{H}$  and a relation  $R \in R_\lambda$  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  $\phi, x, s, s', \omega$  with corresponding index sets  $\Phi, X, S, S', \Omega$  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 write  $a \in R$  or alternatively  $(\phi, x, s, s', \omega) \in R$  to denote that the vector  $a$  satisfies relation  $R$ . We say that the sequence of  $r$  variable vectors  $((\phi_1, x_1, s_1, s'_1, \omega_1), \dots, (\phi_r, x_r, s_r, s'_r, \omega_r)) \in R^r$  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  $\text{pp}$ . In order to relate corresponding state and state update pairs, we require a bijective map  $s2s$  between index sets for states  $s2s : S \rightarrow S'$ . If state variables are ordered appropriately, this can be as simple as a fixed offset.

An ADSC-SNARK for  $R_\lambda$  consists of the following tuple of algorithms ( $\text{Gen}$ ,  $\text{Setup}$ ,  $\text{Auth}$ ,  $\text{Prove}$ ,  $\text{Verify}$ ).

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

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

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

**Algorithm**  $(\pi_t, c_t, p_t) \leftarrow \text{Prove}(\sigma_p, a_t, \nu_t, p_{t-1})$ : gets prover-key  $\sigma_p$ , variables  $a_t = (\phi_t, x_t, s_t, s'_t, \omega_t)$ , and signature  $\nu_t$ . It outputs a proof  $\pi_t$  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**  $b \leftarrow \text{Verify}(\sigma_v, \phi_t, \pi_t, c_t, c_{t-1}, t)$ : gets verifier-key  $\sigma_v$ , the public input-output  $\phi_t$ , the proof  $\pi_t$ , 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 (non-deterministic) computation of function  $f$  such that  $R$  is satisfied for  $(\phi_t, x_t, s_t, s'_t, \omega_t)$ , when  $(y_t, s'_t) = f(x_t, s_t)$ . To represent state, we have 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  $y_t, s_{t+1} = f(x_t, s_t)$ .

The public input-output  $\phi$  is the part of the variables sent to  $\mathcal{V}$ . In our example  $\phi$  is the function's output  $y$ . Private input  $x$  corresponds to the function's input  $x_t$  for iteration  $t$ . Note that it is *not* sent to the verifier.

Finally, witness  $\omega$  is a tuple of variables that appear in the relation and can be seen as non-deterministic advice for the computation of  $f$ .

## 2.2 ADSC-SNARK Properties

For an ADSC-SNARK we require completeness, knowledge-soundness, and succinctness. 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  $\lambda \in \mathbb{N}$ ,  $\text{pp} \leftarrow \text{Gen}(\lambda)$ ,  $(a_1, \dots, a_r) \in \mathbb{R}^r$ , 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  $p_0 = \{\}$ :

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

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  $\text{Auth}$ .

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

that

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

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

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. For authenticity, the definition captures chosen message attacks against the authenticating party by giving  $\mathcal{A}$  oracle access to  $\text{Auth}$ .

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  $T_{\text{Verify}}$  of Algorithm  $\text{Verify}$  are even constant in the length of witness  $\omega$ , the length of input  $x$ , the size of states  $s$ . The verifier runtime is linear only in the size of statement  $\phi$ . We formalize this with the following definition.

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

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

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

### 2.3 ADSC-SNARK Relation

Relation  $\mathbb{R}$  in our ADSC-SNARK construction will be given as a Quadratic-Arithmetic-Program (QAP). QAPs have turned out to be useful in practice, as there exist straight-forward reductions from Arithmetic Circuits (see [4]) and from Rank-1-Constraint-Systems (R1CS), a popular type of relation to represent computations [31].

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  $\Phi, X, S, S', \Omega$ , 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

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

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 ([20–22]) 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. [32]), 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 Main ADSC-SNARK Construction

Before presenting our main contribution, the construction of ADSC-SNARK, we start with a more formal overview and highlight the main ideas to ease understanding.

#### 3.1 Overview

Our ADSC-SNARK takes the SNARK of Groth [7] 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 their prover key  $\sigma_p$ . Informally, the security rationale is that the prover cannot create a valid proof besides by linearly combining elements from  $\sigma_p$ , as the verifier will check certain relations between the proof elements that depend on a set of secret values  $(\alpha, \beta, \gamma, \dots)$  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 the prover key elements that depend on the secret values, but do not reveal them. In fact, the prover must use variables  $(\phi, x, s, s', \omega) \in \mathbb{R}$  as coefficients of these linear combinations for a proof to verify. Recall that  $(\phi, x, s, s', \omega) \in \mathbb{R}$  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 [20, 21, 33, 34]: 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 [35]. 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 committed 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  $\pi_{t-1}$  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  $\pi_t$  (and also another commitment  $c_t$ ). Now, in this iteration  $t$ , the verifier can not only check whether  $\pi_t$  is correct, but also whether state  $s_t$  committed to by some of the elements in  $\pi_t$  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:** On a high level, we use a similar technique as for state consistency to also enable proving data authenticity by letting 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  $\pi_t$  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:** To actually verify that two SNARK proofs or a SNARK proof and a commitment have been computed using the same data, there exist two main approaches in the literature: Costello et al. [20] split some of the proof elements of the BCTV14 SNARK [6] into two parts, one part for the committed variables and the other part for the remaining variables used in the proof. Now, the commitment part can be combined with proofs of the same relation, but with different remaining variables. This ensures that the committed variables are the same across the different proofs. The alternative by Campanelli et al. [21] is to add an additional intermediate commitment for their modular LegoSNARK framework that not only allows to connect proofs of the same relation, but also for different relations and even different SNARKs and commitment schemes. This approach requires an additional strengthening step of the QAP (increasing its degree to make polynomials linearly independent) for the prover key elements to be sufficiently “random” which results in an arithmetization overhead.

In ADSC-SNARKs, we design a novel strategy for checking proofs and commitments for consistency that does not require the strengthening step of the QAP and reduces the required number of proof elements. For commitments  $[D_t]_1$  and  $[E_t]_1$ , we let **Setup** generate random group elements from  $\mathbb{G}_1$ . 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  $\left\{ \frac{P_i(z) - R_{s2si}}{\delta} \right\}_{i \in S}$

$\left\{ \frac{P_i(z) - R_i}{\delta} \right\}_{i \in S'}, \left\{ \frac{P_i(z) - T_i}{\delta} \right\}_{i \in X}$ ). 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  $\varepsilon, \kappa, \eta$  to prevent the prover from mixing coefficients across the different commitments.

**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  $\eta R_i, \kappa R_i$ , and  $\varepsilon T_i$  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 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

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

We begin with the technical details of our ADSC-SNARK construction. For now, we ignore prover state  $p_i$ , 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.

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

Setup comprises two parts:

1.  $(\sigma_a, \sigma_{aux}) \leftarrow \text{Setup}_a()$ :  
Pick  $T_i \leftarrow_{\$} \mathbb{F}_p^*$  for  $i \in X$ .  
Compute  $(sk, pk) \leftarrow \text{SigGen}()$ .  
Set  $\sigma_a = (sk, \{[T_i]_1\}_{i \in X})$ ,  $\sigma_{aux} = (\{[T_i]_1\}_{i \in X}, pk)$ .

2.  $(\sigma_p, \sigma_v, c_0) \leftarrow \text{Setup}_{\text{pv}}(s'_0, \sigma_{aux})$ :  
 Parse  $\{[T_i]_1\}_{i \in X}$ ,  $pk$  from  $\sigma_{aux}$ .  
 Pick  $\alpha, \beta, \gamma, \delta, \kappa, \eta, \varepsilon, z \leftarrow \mathbb{F}_p^*$ ,  $R_i \leftarrow \mathbb{F}_p^*$  for  $i \in S'$ .  
 Compute

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

$$\sigma_{p,2} = (\beta, \{v_i(z)\}_{i \in [0,m]})$$

$$\sigma_{v,1} = \left( \left\{ \frac{P_i(z)}{\gamma} \right\}_{i \in \{0\} \cup \Phi} \right)$$

$$\sigma_{v,2} = (\gamma, \delta, \varepsilon, \eta, \kappa).$$

Set  $\sigma_p = ([\sigma_{p,1}]_1, [\sigma_{p,2}]_2)$ ,  
 and  $\sigma_v = ([\sigma_{v,1}]_1, [\sigma_{v,2}]_2, [\alpha\beta]_T, pk)$ .  
 Compute  $c_0 = [E_0]_1 = \sum_{i \in S'} s'_{0, \text{pos}_{S'}(i)} [R_i]_1$ .

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

Parse  $sk, \{[T_i]_1\}_{i \in X}$  from  $\sigma_a$ .  
 Compute  $\nu_t = ([D]_1, sig)$   
 with  $D = \sum_{i \in X} a_i T_i$ , and  $sig = \text{SigSign}(sk, ([D]_1, t))$ .

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

Parse  $[D]_1, sig$  from  $\nu_t$ .  
 Compute the coefficients of polynomial  $h(Z)$  by polynomial division:

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

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

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

$$C = \sum_{i \in \Omega} \frac{a_i P_i(z)}{\delta} + \sum_{i \in S} \frac{a_i (P_i(z) - \eta R_{s2s(i)})}{\delta}$$

$$+ \sum_{i \in S'} \frac{a_i (P_i(z) - \kappa R_i)}{\delta} + \sum_{i \in X} \frac{a_i (P_i(z) - \varepsilon T_i)}{\delta} + \frac{h(z)t(z)}{\delta},$$

$$E = \sum_{i \in S'} a_i R_i.$$

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

After parsing  $[A_t]_1, [B_t]_2, [C_t]_1, [D_t]_1, sig_t$  from  $\pi_t$ ,  
 setting  $[E_t]_1 = c_t$  and  $[E_{t-1}]_1 = c_{t-1}$  and parsing  $a_{t,i}$  from  $\phi_t$  for  $i \in \Phi$  and  
 $a_{t,0} = 1$ ,  
 accept proof *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

$$\mathbf{SigVerify}(pk, sig_t, ([D_t]_1, t)) = 1.$$

### 3.3 Security Analysis and Zero-Knowledge

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

### 3.4 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 [7] 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 [7], 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  $\sigma_p$ , and letting the verifier add  $c$  directly to the verification equation is a new efficient method to check, whether committed variables for commitment  $c$  match the variables for proof  $\pi$ . It does not require an additional strengthening of the QAP and does not require additional intermediate proof elements.

### 3.5 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,

$\text{Setup}_a$  produces multiple key pairs for the signature scheme:

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

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

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

and sends it to the prover.

The prover includes all  $[D_j]_1, sig_j$  from  $\nu_{j,t}$  in proof  $\pi_t$ .

The verifier checks the signatures of every  $[D_j]_1$ :

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

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.

### 3.6 Relation Independent Authentication

A property of related constructions such as AD-SNARK [22] or SPHinx [23] is that the authentication is independent of the relation  $R$ , such that data can be authenticated, before a relation is decided. This is also the case with our construction: If  $R$  has not been determined yet, but one wants to already authenticate data, one can let a trusted party run  $\text{Setup}_a$  to obtain authentication key  $\sigma_a$  and auxiliary information  $\sigma_{aux}$ . Note that  $\text{Setup}_a$  does not require any inputs related to  $R$ . The trusted party then stores  $\sigma_{aux}$  for setting up  $\sigma_p$  and  $\sigma_v$  for  $R$  at a later point in time.

## 4 Evaluation

We compare the performance of the full zero-knowledge version of our ADSC-SNARK (see Appendix B.2) against the folklore (“strawman”) approach where a regular SNARK is expanded with authenticated input data and state-consistency by arithmetizing a signature verification algorithm as well as a hash function. 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 SNARKs on authenticated data [22] and, second, a construction that we name *LegoGro SC-SNARK*. The latter is a composition of two LegoGro16 SNARKs from the LegoSNARK framework [21] 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<sup>2</sup>. Libsnark already contains an implementation of AD-SNARK.

**ADSC-SNARK** We have implemented the zero-knowledge version of our ADSC-SNARK. We instantiate the signature scheme by the OpenSSL implementation of EdDSA.

**Strawman ADSC-SNARK** The strawman SNARK for relation  $R$  is a Groth16 SNARK for an augmented relation  $R_a = R \wedge R_x \wedge R_s \wedge R_{s'} \wedge R_i$ . Here,  $R_x$  encodes a signature verification for the private input  $x$ . Relations  $R_s$  and  $R_{s'}$  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 inputs-outputs of the Groth16 SNARK and are considered part of the proof of the 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 [22]), 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<sup>3</sup> by instantiating it with the SNARK-friendly Poseidon hash described by Grassi et al. [36]. 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

<sup>2</sup> <https://github.com/scipr-lab/libsnark>

<sup>3</sup> <https://github.com/HarryR/ethsnarks>

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  $R_{s'}$ , 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.

**AD-SNARK [22]** Backes et al. provide a libsnark implementation of their AD-SNARK construction, supporting both a designated and a public verifier version. We benchmark against both versions.

**LegoGro SC-SNARK** The LegoSNARK framework by Campanelli et al. [21] 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 primitive 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.

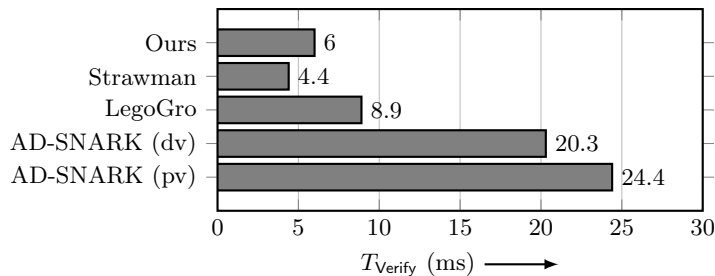
## 4.2 Benchmarks

We have run the benchmarks with the BN-254 elliptic curve which is a standard choice for pairing-based SNARKs. We have instrumented the code with timing functions provided by libsnark that use the per-process CPU timer. We have compiled the code with the GCC C++ compiler set to 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 have not used libsnark’s multiprocessing capabilities and ran the benchmarks within a single thread for better reproducibility. For each data point, we have constructed a basic relation with  $2^{15}$  constraints as well as the provided number of inputs and states. The size of the basic relation corresponds to a medium sized relation.

We have evaluated our ADSC-SNARK as well as LegoGro SC-SNARK and AD-SNARK for 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. 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. This allows to compare against the implementation of Backes’ AD-SNARK, as their implementation does not support public input-outputs additional to private inputs without modifying the code.

**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 security properties as ADSC-SNARKs, see Table 1.



**Fig. 1.** 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.

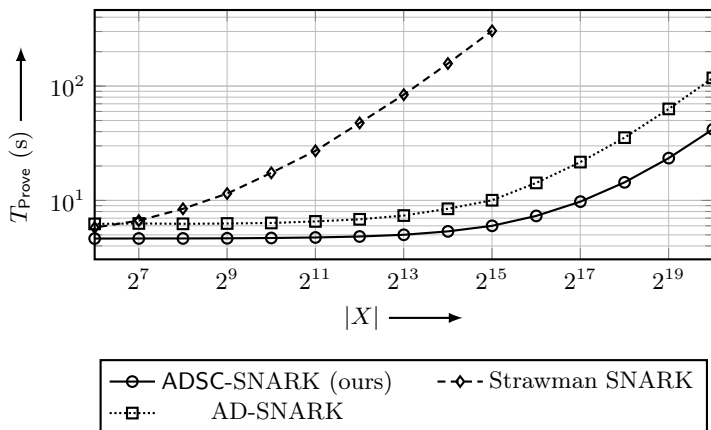
Figure 1 compares  $T_{\text{Verify}}$  of the different SNARKs. The verifier runtimes of our ADSC-SNARK, the strawman SNARK, and LegoGro SC-SNARK are constant. For AD-SNARK, the plot shows the runtime with  $|X| = 1$  private inputs. Compared to AD-SNARK, our SNARK verifier needs 70% (designated verifier) 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,  $T_{\text{Verify}}$  is 101.4ms for the designated verifier version and 34.9s 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 3 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

**Table 3.** Proof size

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

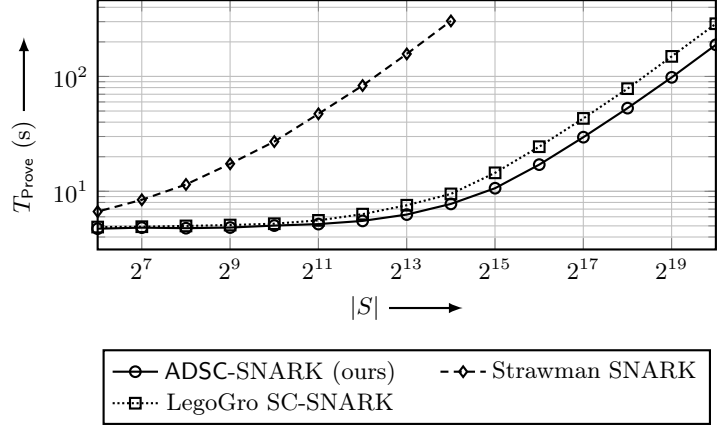
serialization. The serialization routines compress elliptic curve points by storing just one coordinate and a sign bit. However, there is additional overhead for paddings to align objects to 8-bit data chunks and for additional separation tokens between data objects. While the strawman SNARK achieves 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.



**Fig. 2.** Prover runtime, varying private inputs. AD-SNARK does not provide state consistency.

Figure 2 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  $29\times$  faster than the strawman SNARK and  $1.6\times$  faster than AD-SNARK. Note, that AD-SNARK does not support state-consistency. Figure 2 does not include LegoGro SC-SNARK, as it does not support private inputs.

Figure 3 compares the prover runtime with respect to the number of states. For  $2^{14}$  states, our zero-knowledge ADSC-SNARK is  $39\times$  faster than the strawman SNARK and  $1.2\times$  faster than LegoGro SC-SNARK. Figure 3 does not include AD-SNARK, as it does not support state consistency.



**Fig. 3.** Prover runtime, varying states. LegoGro SC-SNARK does not provide authenticated data.

**Table 4.** 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 $	$2^0$			$2^7$			$2^{14}$		
	Strawman	Ours	Speedup	Strawman	Ours	Speedup	Strawman	Ours	Speedup
$2^0$	0.64 s	0.25 s	$\times 2.5$	3.80 s	0.32 s	$\times 11.9$	303 s	4.38 s	$\times 69$
$2^7$	2.46 s	0.27 s	$\times 9.1$	5.05 s	0.33 s	$\times 15$	303 s	4.30 s	$\times 70$
$2^{14}$	154 s	1.04 s	$\times 147$	155 s	1.11 s	$\times 140$	447 s	5.03 s	$\times 89$

Table 4 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, up to about a factor of 150.

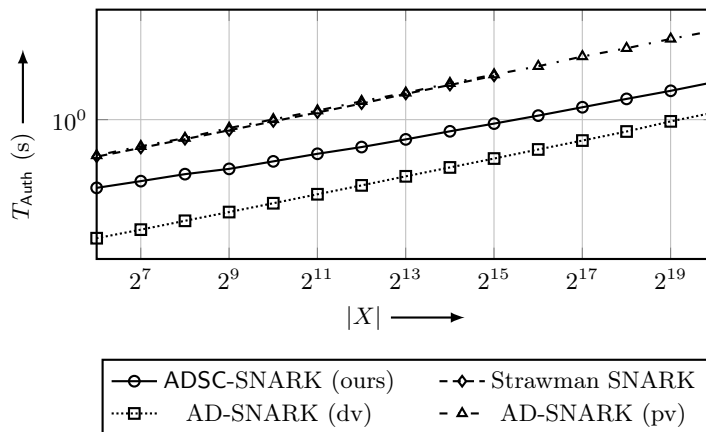


Fig. 4. Authenticator runtime. AD-SNARK does not provide state consistency.

Figure 4 compares the authenticator runtimes of our zero-knowledge ADSC-SNARK, Backes et al.’s AD-SNARK with the public verifier (pv) version and the designated verifier (dv) version, as well as the strawman SNARK. The runtime of the strawman SNARK and the public verifier AD-SNARK are the largest and overlap in most of the plot. The designated verifier AD-SNARK is the fastest, our ADSC-SNARK is in between. Note, however, that our ADSC-SNARK is publicly verifiable in contrast to the faster AD-SNARK version.

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 [22] and LegoGro SC-SNARK due to the optimizations presented earlier.

## 5 Applications

Many real-world computations are both stateful and iterative. ADSC-SNARKs can be used to make outputs of such a computation verifiable, even if the computational capabilities and communication bandwidth of the verifying party are

limited. Our motivating application is a digital control system. To demonstrate the feasibility of applying ADSC-SNARK in this setting, we arithmetize a cybernetic control law for a commercial quadrotor (“quadcopter”) UAV attitude stabilization system and setup a simulation for evaluation.

## 5.1 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. Prominent examples are flight control in aircraft, electrical power grid control, power generation plant control or control in automated industrial processes.

Due to the complexity of these systems, there often exist trust boundaries or no trust between components. For example, in a modern commercial aircraft different suppliers provide sensors, electronic control units, and actuators [37]. Also, the systems’ complexity and their interconnectivity results in a large attack surface. Yet, many of these applications are safety critical, i.e., a malfunction or adversarial tampering can lead to loss of lives.

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 [38]. 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 [27]. An ADSC-SNARK would fit this application particularly well: sensors would authenticate their measurements, the control unit provides in each iteration control outputs together with a proof of output correctness, and actuators verify that outputs have been correctly computed. In case of a detected malfunction, the system could then perform graceful degradation (such as automated landing) or require manual intervention.

**Simulation** To evaluate the feasibility of applying ADSC-SNARKs in such a setting, we set up a simulation of four sensors, a control unit, and an actuation unit. This setup reflects a standard quadrotor flight control system and is shown in Figure 5. The four sensors measure 6 physical properties. Every physical property is measured twice to cope with sensor failures. Sensors 1 and 2 both measure the state of an input device, e.g., the deflection of a control stick in three axes  $c_x$ ,  $c_y$ , and  $c_z$ . Sensors 3 and 4 measure the physical state of the system, e.g., the attitude of a quadrotor in Euler angles  $\theta_x$ ,  $\theta_y$ , and  $\theta_z$ . They periodically send messages containing the measurements and corresponding ADSC-SNARK’s

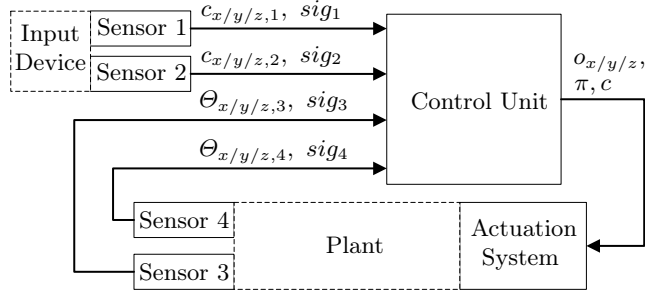


Fig. 5. Simulation scenario of a verifiable control system for quadrotor stabilization.

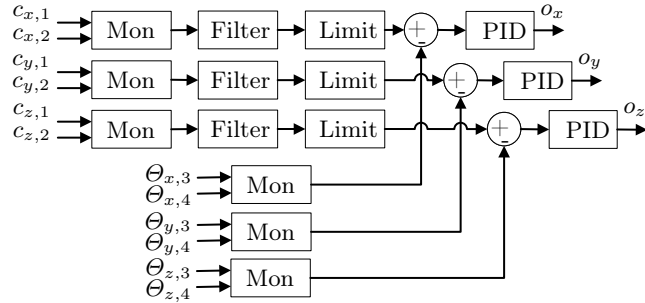


Fig. 6. Quadrotor attitude stabilization control law, 3-dimensional PID controller

message signatures  $sig$  to the (untrusted) control unit. Sensors each have their own authentication key as described in Section 3.5. The control unit applies a cyber-physical control law to compute outputs  $o_x, o_y$ , and  $o_z$  and it generates a proof  $\pi$  that is sent to the actuation system along with the outputs. The actuation system uses ADSC-SNARK’s verification algorithm to check the outputs in each iteration. The plant in Figure 5 represents the physics of the system to be controlled, in this case the dynamics and kinematics of the quadrotor. As this part is not relevant for our evaluation, it is not realized in the simulation.

**Control Law** The specific control law evaluated and proven by the control unit is shown as a block diagram in Figure 6. It comprises the standard elements of attitude control laws for quadcopters [39–41], including a three-degrees of freedom PID controller and various signal filtering and shaping elements.

First, for each input signal, a monitoring function  $Mon$  is applied.  $Mon$  checks that the difference of redundant input signals is within a defined limit and computes an average. This is a common technique in safety-critical systems to ensure correctness of sensor inputs. If input signals differ by too much, for example because one of the hardware sensors provides erroneous signals, the control unit will not be able to produce a verifying proof.



The input signals are further processed by a first-order low-pass *Filter*. This is also a common technique to attenuate high-frequency noise from a signal in control systems. Each *Filter* has one state. In the next step, the filtered signal is constrained by a set of bounds. The output of *Limit* is the input value, if it lies within the bounds, or the limit value, if it is outside the bounds.

Finally, the difference of the limited command and the measured signal is given as an input to *PID*, a discrete Proportional-Integral-Derivative filter, which has two states. In total, the controller has 12 inputs, 3 outputs, and 9 states. The relation representing the evaluation of this control law is expressed in R1CS with 1065 constraints.

**Benchmarks** 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 system to verify the command 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

If the untrusted control unit is expected to not behave maliciously, but hardware faults occur statistically, e.g., randomly with a certain probability, one can significantly reduce the security parameter of SNARKs [27]. Consequently, we present benchmarks for a choice of different elliptic curves and security parameters:

- BLS12-381: A Barreto-Lynn-Scott elliptic curve [42] with embedding degree 12 and prime group order of 255 bits,
- BN254, BN183, BN124: Barreto-Naehring elliptic curves [43] with embedding degree 12 and 254, 183, and 124 bits prime group order,
- GMV181, GMV97, GMV58: Galbraith-McKee-Valença elliptic curves [44] with embedding degree 6 and 181, 97, and 58 bits prime group order.

Curve BLS12-381 is the elliptic curve used for the ZCash cryptocurrency and specified in [45]. BN254 was constructed by Ben-Sasson et al. [6] and is the default curve in `libsnark`, also referred to as `alt_bn128`. GMV181 is the curve used by Ben-Sasson et al. [5]. We constructed and implemented the other curves according to the methods of Barreto and Naehrig as well as Galbraith et al.. The curve parameters and implementations can be found in the accompanying source code. Table 5 shows runtime measurements and the communication overhead for sensors (authentication tag size) and control unit (proof size). The simulation

**Table 5.** Runtime and communication of digital control system demo. Proof size includes proof and commitment.

Elliptic Curve	Control Unit runtime	Actuation System runtime	Proof size
BLS12-381	165.8 ms	12.0 ms	712 B
BN254	82.3 ms	6.9 ms	568 B
BN183	39.5 ms	3.7 ms	496 B
BN124	35.2 ms	3.3 ms	442 B
GMV181	34.2 ms	3.2 ms	512 B
GMV97	24.9 ms	2.5 ms	432 B
GMV58	13.8 ms	1.5 ms	352 B

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 65 Hz update frequency. Again, we stress that smaller prime group curves do not provide security against a 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 [46]) and 100 Hz (Boeing Fly-by-Wire system [47]) and small quadrotors have update rates from 50 Hz up to several kHz [41]. 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.

## 5.2 Further Applications

We briefly present additional applications for ADSC-SNARKs to show the usefulness of our construction.

**Trustless Audio and Video Streaming** With the rapid increase of photo-realism and quality of machine generated audio and video, it will be hard to distinguish artificial from actual video and audio footage leading to fake news and targeted misinformation campaigns. This could be tackled by letting the capturing hardware sign audio or video streams. If the stream, however, is processed by compression methods or by applying filtering, such a signature cannot be verified anymore. Such compression methods or filters can be stateful, as their

outputs might depend not only on the current, but on previous frames of the stream. ADSC-SNARK can solve this problem by enabling proofs, that signed and streamed data has undergone a set of allowed transformations, and therefore make the final result traceable to the recording hardware. Compared to previous folklore approaches that perform expensive signature and hash function evaluations within the circuits, our ADSC-SNARK significantly reduce the overhead in this already computationally intensive application.

**Stateful Machine Learning** Some deep neural network types including recurrent neural networks and long short-term memory networks are stateful. Computationally expensive instances usually do not directly run on consumer’s hardware. Instead, they are outsourced to server centers. In such a setting, a user might wish to verify, that the correct model was applied to its query without needing to trust the service provider. Modern machine learning models can also take external data as an input and the user might want to be asserted, that this data originates from a trusted source. In such a case, ADSC-SNARKs make model outputs verifiable without disclosing inputs while at the same time reducing the additional overhead introduced by folklore approaches.

## 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 [20, 25] or parts thereof [26]. The first SNARK-based solution that does not require arithmetizing a signature verification algorithm is AD-SNARK by Backes et al. [22]. 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 [23] 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 [24] 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 [48–50] and their publicly verifiable counterpart: homomorphic signatures [51, 52]. However, these constructions

are either concretely inefficient as they build on fully homomorphic encryption ([48]), or on cryptographic multi-linear maps ([50, 52]), or they are not succinct ([49, 51]). 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 [28] or more complex structures such as Merkle-trees [16, 29] or RSA accumulators [30]. These approaches significantly increase the size of the relation and therefore prover runtime.

Gepetto [20] introduces an approach that does not increase the size of the relation. It extends the Pinocchio SNARK [4] to a commit-and-prove scheme, enabling proofs on relations with a shared state. Campanelli et al. [21] 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 [33, 34, 53].

A somewhat orthogonal approach to the idea of stateful proofs is incrementally verifiable computation (IVC) [17, 54, 55] 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 many real-world applications including control systems, but also in various streaming, or machine learning applications.

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  $89\times$  reduction in prover time. Compared to related, more efficient, approaches, ADSC-SNARK achieve better prover time, verification time and smaller proof size.

## Bibliography

- [1] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 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*, ACM Conferences, pages 326–349. ACM, 2012.
- [2] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct Non-interactive Arguments via Linear Interactive Proofs. In *Theory of Cryptography: 10th Theory of Cryptography Conference*, pages 315–333. Springer, Berlin, Heidelberg, 2013.
- [3] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In *Advances in Cryptology – EUROCRYPT 2013*, pages 626–645. Springer, Berlin, Heidelberg, 2013.
- [4] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013.
- [5] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *Proceedings of the 33rd Annual International Cryptology Conference*, pages 90–108. Springer, Berlin, Heidelberg, 2013.
- [6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *23rd USENIX Security Symposium*, pages 781–796. USENIX Association, 2014.
- [7] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326. Springer Berlin Heidelberg, 2016.
- [8] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkatasubramanian. Liger: Lightweight Sublinear Arguments Without a Trusted Setup. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2087–2104. ACM, 2017.
- [9] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In Lorenzo Cavallaro, editor, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ACM Digital Library, pages 2111–2128. Association for Computing Machinery, 2019.
- [10] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *Cryptology ePrint Archive*, pages 1–34, 2019.
- [11] Srinath Setty. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In Daniele Micciancio and Thomas Ristenpart, editors,

- Advances in Cryptology – CRYPTO 2020*, volume 12172 of *Springer eBook Collection*, pages 704–737. Springer International Publishing and Imprint Springer, 2020.
- [12] Srinath Setty and Jonathan Lee. Quarks: Quadruple-efficient transparent zkSNARKs. *Cryptology ePrint Archive*, 2020.
  - [13] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In Anne Canteau and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, Lecture Notes in Computer Science, pages 738–768. Springer, Cham, 2020.
  - [14] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent SNARKs from DARK Compilers. In Anne Canteau and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, Springer eBook Collection, pages 677–706. Springer International Publishing and Imprint Springer, 2020.
  - [15] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-Time and Field-Agnostic SNARKs for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, Lecture Notes in Computer Science, pages 193–226. Springer Nature Switzerland and Imprint Springer, 2023.
  - [16] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Scalable Zero Knowledge via Cycles of Elliptic Curves. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology – CRYPTO 2014*, volume 8617 of *Lecture Notes in Computer Science*, pages 276–294. Springer Berlin Heidelberg, 2014.
  - [17] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive Zero-Knowledge Arguments from Folding Schemes. In *Advances in Cryptology - CRYPTO 2022*, pages 359–388. Springer, Cham, 2022.
  - [18] Abhiram Kothapalli and Srinath Setty. HyperNova: Recursive Arguments for Customizable Constraint Systems. In Leonid Reyzin and Douglas Stebila, editors, *Advances in Cryptology – CRYPTO 2024*, Lecture Notes in Computer Science, pages 345–379. Springer Nature Switzerland and Imprint Springer, 2024.
  - [19] Liam Eagen and Ariel Gabizon. ProtoGalaxy: Efficient ProtoStar-style folding of multiple instances. *Cryptology ePrint Archive*, pages 1–18, 2023.
  - [20] Craig Costello, Cedric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile Verifiable Computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE, 2015.
  - [21] Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2075–2092. ACM, 2019.
  - [22] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. ADSNARK: Nearly Practical and Privacy-Preserving Proofs on Authenticated Data. In *2015 IEEE Symposium on Security and Privacy*, pages 271–286. IEEE, 2015.

- [23] Dario Fiore and Ida Tucker. 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.
- [24] Trisha Datta, Binyi Chen, and Dan Boneh. VerITAS: Verifying Image Transformations at Scale. *Cryptology ePrint Archive*, 2024.
- [25] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, T-H. Hubert Chan, Charalampos Papamanthou, Rafael Pass, Abhi Shelat, and Elaine Shi. C0C0: A Framework for Building Composable Zero-Knowledge Proofs. *Cryptology ePrint Archive*, 2015.
- [26] Zhiguo Wan, Zhangshuang Guan, Yan Zhou, and Kui Ren. zk-AuthFeed: How to Feed Authenticated Data into Smart Contract with Zero Knowledge. In *2019 IEEE International Conference on Blockchain*, pages 83–90. IEEE, 2019.
- [27] Johannes Reinhart, Bastian Luettig, Nicolas Huber, Julian Liedtke, and Bjoern Annighofer. Verifiable Computing in Avionics for Assuring Computer-Integrity without Replication. In *Digital Avionics Systems Conference 2023*, pages 1–10, 2023.
- [28] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 341–357. ACM, 2013.
- [29] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. Fast reductions from RAMs to delegatable succinct constraint satisfaction problems. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 401–413. ACM Press, 2013.
- [30] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling Verifiable Computation Using Efficient Set Accumulators. In *Proceedings of the 29th USENIX Security Symposium*, pages 2075–2092. USENIX Association, 2020.
- [31] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A Circuit Description Language for Building Zero-Knowledge Applications. *IEEE Transactions on Dependable and Secure Computing*, 20(6):4733–4751, 2023.
- [32] Bryan Parno. A Note on the Unsoundness of vnTinyRAM’s SNARK. *Cryptology ePrint Archive*, pages 1–4, 2015.
- [33] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash First, Argue Later. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1304–1316. ACM, 2016.
- [34] Helger Lipmaa. Prover-Efficient Commit-and-Prove Zero-Knowledge SNARKs. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in cryptology - AFRICACRYPT 2016*, Lecture notes in computer science Security and cryptology, pages 185–206. Springer, 2016.

- [35] Torben Pryds Pedersen. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, SpringerLink Bücher, pages 129–140. Springer-Verlag Berlin Heidelberg, 1992.
- [36] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *Proceedings of the 30th USENIX Security Symposium*, pages 519–535. USENIX Association, 2021.
- [37] NBC. Hundreds of suppliers, one Boeing 737 airplane, 2010. <https://www.nbcnews.com/id/wbna36507420>.
- [38] Kevin Driscoll, Brendan Hall, Michael Paulitsch, Phil Zumsteg, and Håkan Sivencrona. The real Byzantine Generals. In *The 23rd Digital Avionics Systems Conference*, pages 6.D.4–61–11. IEEE Operations Center, 2004.
- [39] Atheer L. Salih, M. Moghavvemi, Haider A. F. Mohamed, and Khalaf Salom Gaeid. Modelling and PID controller design for a quadrotor unmanned air vehicle. In *2010 IEEE International Conference on Automation, Quality and Testing, Robotics*, pages 1–5. IEEE, 2010.
- [40] Jun Li and Yuntang Li. Dynamic analysis and PID control for a quadrotor. In *2011 IEEE International Conference on Mechatronics and Automation*. IEEE, 2011.
- [41] José Claudio Vianna Junior, Julio Cesar de Paula, Gideon Villar Leandro, and Marlio Couto Bonfim. Stability Control of a Quad-rotor Using a PID Controller. *Brazilian Journal of Instrumentation and Control*, 1(1):15, 2013.
- [42] Barreto, Paulo S. L. M., Ben Lynn, and Michael Scott. Constructing Elliptic Curves with Prescribed Embedding Degrees. In *International Conference on Security in Communication Networks*, pages 257–267. Springer, Berlin, Heidelberg, 2003.
- [43] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In *International workshop on selected areas in cryptography*, pages 319–331. Springer, Berlin, Heidelberg, 2005.
- [44] S. D. Galbraith, J. F. McKee, and P. C. Valença. Ordinary abelian varieties having small embedding degree. *Finite Fields and Their Applications*, 13(4):800–814, 2007.
- [45] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. ZCash Protocol Specification: Version 2022.3.8 [NU5], 15.09.2022. URL <https://zips.z.cash/protocol/protocol.pdf>.
- [46] G. M. Minott, J. B. Peller, and K. J. Cox. Space Shuttle Digital Flight Control System. *Advanced Control Technology and its Potential for Future Transport Aircraft*, pages 271–294, 1976.
- [47] Robert Bleeg. Commercial jet transport fly-by-wire architecture considerations. In *Digital Avionics Systems Conference*, pages 399–406. American Institute of Aeronautics and Astronautics, 1988.
- [48] Rosario Gennaro and Daniel Wichs. Fully Homomorphic Message Authenticators. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, Lecture Notes in Computer Science, pages 301–320. Springer, 2013.



- [49] Dario Catalano and Dario Fiore. Practical Homomorphic MACs for Arithmetic Circuits. In *Advances in Cryptology - EUROCRYPT 2013*, pages 336–352. Springer, Berlin, Heidelberg, 2013.
- [50] Dario Catalano, Dario Fiore, Rosario Gennaro, and Luca Nizzardo. Generalizing Homomorphic MACs for Arithmetic Circuits. In Hugo Krawczyk, editor, *Public-key cryptography*, Lecture notes in computer science Security and cryptology, pages 538–555. Springer, 2014.
- [51] Dan Boneh and David Mandell Freeman. Homomorphic Signatures for Polynomial Functions. In *Advances in Cryptology - EUROCRYPT 2011*, pages 149–168. Springer, Berlin, Heidelberg, 2011.
- [52] Dario Catalano, Dario Fiore, and Bogdan Warinschi. Homomorphic Signatures with Efficient Verification for Polynomial Functions. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014*, Lecture notes in computer science Security and cryptology, pages 371–389. Springer, 2014.
- [53] Meilof Veeningen. Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation. In Marc Joye and Abderrahmane Nitaj, editors, *Progress in Cryptology - AFRICACRYPT 2017*, Lecture notes in computer science Security and cryptology, pages 21–39. Springer, 2017.
- [54] Paul Valiant. Incrementally Verifiable Computation or Proofs of Knowledge Imply Time/Space Efficiency. In *Theory of Cryptography: Fifth Theory of Cryptography Conference*, pages 1–18. Springer, Berlin, Heidelberg, 2008.
- [55] Omer Paneth and Rafael Pass. Incrementally Verifiable Computation via Rate-1 Batch Arguments. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science*, pages 1045–1056. IEEE, 2022.

## 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 [7].

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  $A_t, B_t, C_t, \dots$  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  $\sigma_p$ , as Equation 2 checks a relation depending on secret values  $\alpha, \beta, z, \dots$ . 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  $\sigma_p$ , 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[7]).

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  $(sk, pk) \leftarrow \text{SigGen}(\lambda)$ , signing algorithm  $sig \leftarrow \text{SigSign}(sk, m)$  and verification algorithm  $b \leftarrow \text{SigVerify}(pk, sig, m)$  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  $\lambda$  and messages  $m$ :

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

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

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

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:**  $\sigma_c \leftarrow \text{PedSetup}(\lambda)$ :

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:**  $(c, o) \leftarrow \text{PedCommit}(\sigma_c, v)$ :

Parse  $T_i$  from  $\sigma_c$ .

Pick:  $o \leftarrow_{\$} \mathbb{F}_p^*$ , Compute  $c = o \cdot T_0 + \sum_{i \in [1, n]} v_i \cdot T_i$ .

**Verify:**  $b \leftarrow \text{PedVerify}(\sigma_c, c, v, o)$ :

Parse  $T_i$  from  $\sigma_c$ .

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  $\lambda$  and vectors  $v$ :

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

A pedersen vector commitment is binding:

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

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

### 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  $\gamma, \delta, \eta, \kappa$ , and  $\varepsilon$ , 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

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

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:

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

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

**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  $(\sigma_p, \sigma_v)$  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 [7]), the argument in the proof of Theorem 2 in [7]) applies: The reference-string elements can be regarded as multivariate polynomials in  $\alpha, \beta, z, \dots$ , 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  $\alpha, \beta, z, \dots$ . This does not disclose any additional information, as the same test on any other reference-string (with different  $\alpha, \beta, z, \dots$ ) will also

evaluate to zero. 2.) The multivariate polynomials evaluate to zero for a set of specific variables  $\alpha, \beta, z, \dots$ . 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  $\alpha, \beta, \gamma, \dots$ . The only information about the secret values available to  $\mathcal{A}$  are the elements of the reference-string  $(\sigma_p, \sigma_v)$  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  $\alpha, \beta, \gamma, \delta, \kappa, \eta, \varepsilon, z, \{T_i\}_{i \in X}, \{R_i\}_{i \in S'}$  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:

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

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

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

We do the equivalent for  $B$ . We will show later that  $A_\alpha$  and  $B_\beta$  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  $\alpha^2$ , 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  $A_\alpha$  nor  $B_\beta$  can be zero. Thus,  $B'_\alpha = 0$  and  $A_\alpha B_\beta = 1$ . Comparing coefficients of  $\beta^2$ , 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:

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

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

We now have

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

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

$$\begin{aligned} & \alpha \left( \sum_{i \in [0, m]} B'_{(u_i(z))} u_i(z) + \sum_{i \in [0, m]} B'_{(v_i(z))} v_i(z) \right) \\ & = \sum_{i \in \{0\} \cup \Phi} a_{t, i} \cdot \alpha v_i(z) + \sum_{i \in \Omega} C_{t, P_i(z)/\delta} \cdot \alpha v_i(z) \\ & + \sum_{i \in S} C_{t, (P_i(z) - \eta R_{s_{2s}(i)})/\delta} \cdot \alpha v_i(z) \\ & + \sum_{i \in S'} C_{t, (P_i(z) - \kappa R_i)/\delta} \cdot \alpha v_i(z) \\ & + \sum_{i \in X} C_{t, (P_i(z) - \varepsilon T_i)/\delta} \cdot \alpha v_i(z) \end{aligned}$$

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

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

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

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

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

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

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.  $(a_{t,i})_{i \in [1,m]} = (\phi_t, x_t, s_t, s'_t, \omega_t)$ ,  $(\phi_t, x_t, s_t, s'_t, \omega_t) \in \mathbb{R}$ . 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  $\eta R_i$  for each  $i \in S$ , we get:

$$\forall i \in S : E_{t-1, R_{s_{2s(i)}}} \eta R_{s_{2s(i)}} - a_{t,i} \eta R_{s_{2s(i)}} = 0 \Rightarrow \forall i \in S : E_{t-1, R_{s_{2s(i)}}} = a_{t,i}.$$

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

$$\begin{aligned} \forall i \in S : E_{t, R_{s_{2s(i)}}} \kappa R_{s_{2s(i)}} - a_{t,s_{2s(i)}} \eta R_{s_{2s(i)}} &= 0 \\ \Rightarrow \forall i \in S : E_{t, R_{s_{2s(i)}}} &= a_{t,s_{2s(i)}} \\ \Rightarrow \forall i \in S : E_{t-1, R_{s_{2s(i)}}} &= a_{t-1,s_{2s(i)}}. \end{aligned}$$

Combining both results gets us:

$$\forall i \in S : a_{t-1,s_{2s(i)}} = a_{t,i} \Leftrightarrow s'_{t-1} = s_t.$$

*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  $\varepsilon T_i$  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  $\text{Verify} = 1$ , then  $(\phi_t, x_t, s_t, s'_t, \omega_t) \in R$ , and as shown in step 2.5, if  $\text{Verify} = 1$  then  $s_t = s'_{t-1}$ . For knowledge-soundness it remains to be shown that if  $\text{Verify} = 1$ , then the tuple  $(t, x_t)$  must be part of the set of queries to  $\text{Auth}(\cdot)$ :  $(t, x_t) \in \tilde{X}$ . We do so by assuming that  $\text{Extract}$  can extract an  $x_t$ , such that  $(t, x_t) \notin \tilde{X}$  and  $\text{Verify} = 1$ . From that we construct another adversary  $\mathcal{B}$  with oracle access to  $\text{SigSign}$ , 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:

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

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

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

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

Compute

$$(\sigma_p, \sigma_v, c_0) \leftarrow \text{Setup}_{pv}(s'_0, \sigma_{aux}).$$

Compute

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

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

Note, that  $\mathcal{B}^{\text{SigSign}(\cdot)}$  can answer the queries of  $\mathcal{A}^{\text{Auth}(\cdot)}$  to  $\text{Auth}$  by using the information from  $\sigma_{aux}$  and the oracle access to  $\text{SigSign}$ . 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  $\pi_t$  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

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

to hold (case a), or

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

to hold (case b).

Therefore, by construction,  $\text{Pr}_{\mathcal{B}} \approx \text{Pr}_{\mathcal{A}}$ .

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  $\text{Pr}_{\mathcal{C}}$ .

$(c, v, o, \tilde{v}, \tilde{o}) \leftarrow \mathcal{C}(\sigma_c)$  works as follows:

Parse  $\{T_i\}$  from  $\sigma_c$

Compute  $\sigma_{aux}$  as in  $\text{Setup}_a$ , but use  $T_i$  from  $\sigma_c$ , instead of sampling them.

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

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  $\text{SigSign}$ , which tries to break the security of the signature scheme of Definition 5. It's success probability is  $\text{Pr}_{adv}$ .

$(m, sig) \leftarrow \mathcal{D}^{\text{SigSign}(\cdot)}()$  works as follows: Compute

$$\begin{aligned} (\sigma_a, \sigma_{aux}) &\leftarrow \text{Setup}_a() \\ T_0 &\leftarrow \mathbb{F}_p, T_i \leftarrow \text{Parse}(\sigma_a) \\ \sigma_c &= (T_0, \{T_i\}_{i \in X}). \end{aligned}$$

Run  $\mathcal{B}$

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

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:  $\text{Pr}_{\mathcal{C}} \approx 0$ . As we also assume, that the signature scheme is secure (Definition 5),  $\mathcal{D}$  has also negligible chance of success:  $\text{Pr}_{\mathcal{D}} \approx 0$ . Both  $\mathcal{C}$  and  $\mathcal{D}$  having negligible chance of success means that also  $\text{Pr}_{\mathcal{B}} \approx 0$ . It follows that  $\text{Pr}_{\mathcal{A}} \approx \text{Pr}_{\mathcal{B}} \approx 0$ .  $\square$

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

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

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

**Definition 8 (Zero-knowledge of ADSC-SNARK).** For all  $\lambda \in \mathbb{N}$ ,  $\text{pp} \leftarrow \text{Gen}(\lambda)$ , there exists a PPT algorithm  $(\pi_t, c_t) \leftarrow \text{Sim}(\tau, \phi_t, c_{t-1}, t)$ , such that for all  $(a_1, \dots, a_r) \in \mathbb{R}^r$ , 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  $v \leftarrow \mathcal{A}(\sigma_p, \sigma_v, \sigma_a, \tau, \Pi, C)$ :

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

### 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.

We now present the zero-knowledge ADSC-SNARK construction including the zero-knowledge property. Differences to the version without the zero-knowledge property are highlighted.

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

Again, Setup comprises two parts:

1.  $(\sigma_a, \sigma_{aux}) \leftarrow \text{Setup}_a()$ :  
 Pick  $T_i \leftarrow \mathbb{F}_p^*$  for  $i \in X$ ,  $\delta \leftarrow \mathbb{F}_p^*$ .  
 Compute  $(sk, pk) \leftarrow \text{SigGen}()$ .

Set  $\sigma_a = (sk, \{[T_i]_1\}_{i \in X}, [\delta]_1)$ ,  
 $\sigma_{aux} = (\{[T_i]_1\}_{i \in X}, pk, \delta)$ .  
2.  $(\sigma_p, \sigma_v, c_0, \tau) \leftarrow \text{Setup}_{\text{pv}}(s'_0, \sigma_{aux})$ :  
Parse  $\{[T_i]_1\}_{i \in X}, pk, \delta$  from  $\sigma_{aux}$ .  
Pick  $\alpha, \beta, \gamma, \kappa, \eta, \varepsilon, z \leftarrow \mathbb{F}_p^*, R_i \leftarrow \mathbb{F}_p^*$  for  $i \in S'$ .  
Set  $\tau = (\alpha, \beta, \gamma, \delta, \kappa, \eta, \varepsilon, z, \{R_i\}_{i \in S'}, sk)$   
Compute

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

$$\sigma_{p,2} = (\beta, \delta, \{v_i(z)\}_{i \in [0,m]})$$

$$\sigma_{v,1} = \left( \left\{ \frac{P_i(z)}{\gamma} \right\}_{i \in \{0\} \cup \Phi} \right)$$

$$\sigma_{v,2} = (\gamma, \delta, \varepsilon, \eta, \kappa).$$

Set  $\sigma_p = ([\sigma_{p,1}]_1, [\sigma_{p,2}]_2)$ ,  
and  $\sigma_v = ([\sigma_{v,1}]_1, [\sigma_{v,2}]_2, [\alpha\beta]_T, pk)$ .  
Compute  $c_0 = [E_0]_1 = \sum_{i \in S'} s'_{0, \text{pos}_{S'}(i)} [R_i]_1$ .

$\nu_t \leftarrow \text{Auth}(\sigma_a, t, x_t)$   
Parse  $sk, \{[T_i]_1\}_{i \in X}, [\delta]_1$  from  $\sigma_a$ .  
Pick  $b_D \leftarrow \mathbb{F}_p$   
Compute  $\nu_t = ([D]_1, b_D, sig)$   
with  $D = \sum_{i \in X} a_i T_i + b_D \delta$ ,  
and  $sig = \text{SigSign}(sk, ([D]_1, t))$ .

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

Parse  $[D]_1, b_D, sig$  from  $\nu_t$ .  
Compute the coefficients of polynomial  $h(Z)$  by polynomial division:

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

Pick  $b_A, b_B, b_{t,E} \leftarrow \mathbb{F}_p$ .

Set  $b_{t-1,E} = \begin{cases} 0, & \text{if } p_{t-1} = \{\} \\ p_{t-1}, & \text{else} \end{cases}$ ,  $p_t = b_{t,E}$ .

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

$$\begin{aligned} A &= \alpha + \sum_{i \in [0, m]} a_i u_i(z) + b_A \delta, & B &= \beta + \sum_{i \in [0, m]} a_i v_i(z) + b_B \delta, \\ C &= \sum_{i \in \Omega} \frac{a_i P_i(z)}{\delta} + \sum_{i \in S} \frac{a_i (P_i(z) - \eta R_{s2s(i)})}{\delta} \\ &+ \sum_{i \in S'} \frac{a_i (P_i(z) - \kappa R_i)}{\delta} + \sum_{i \in X} \frac{a_i (P_i(z) - \varepsilon T_i)}{\delta} + \frac{h(z)t(z)}{\delta} \\ &+ b_B A + b_A B - b_a b_B \delta - b_D \varepsilon - b_{t-1,E} \eta - b_{t,E} \kappa, \\ E &= \sum_{i \in S'} a_i R_i + b_{t,E} \delta. \end{aligned}$$

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

After parsing  $[A_t]_1, [B_t]_2, [C_t]_1, [D_t]_1, sig_t$  from  $\pi_t$ , setting  $[E_t]_1 = c_t$  and  $[E_{t-1}]_1 = c_{t-1}$  and parsing  $a_{t,i}$  from  $\phi_t$  for  $i \in \Phi$  and  $a_{t,0} = 1$ ,

Accept proof iff

$$\begin{aligned} [A_t]_1 \cdot [B_t]_2 &= [\alpha\beta]_T + \left( \sum_{i \in \{0\} \cup \Phi} a_{t,i} \left[ \frac{P_i(z)}{\gamma} \right]_1 \right) \cdot [\gamma]_2 + [C_t]_1 \cdot [\delta]_2 \\ &+ [D_t]_1 \cdot [\varepsilon]_2 + [E_{t-1}]_1 \cdot [\eta]_2 + [E_t]_1 \cdot [\kappa]_2 \end{aligned}$$

and

$$\text{SigVerify}(pk, sig_t, ([D_t]_1, t)) = 1.$$

### 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.  $\square$

**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:**  $(\pi_t, c_t) \leftarrow \text{Sim}(\tau, \phi_t, c_{t-1}, t)$

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  $\text{sig} = \text{SigSign}(sk, ([D_t]_1, t))$ .

Set  $\pi_t = ([A_t]_1, [B_t]_2, [C_t]_1, [D_t]_1, \text{sig}), c_t = [E_t]_1$ .

*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  $\text{sig}$  was derived from  $(D_t, t)$  using  $\text{SigSign}$  in both the real and simulated proofs.  $\square$

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

*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  $([\beta]_1, [\{v_i(z)\}_{i \in [0, m]}]_1, \dots)$ . 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.  $\square$

**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.  $\square$

## 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-Lynn-Scott elliptic curves

Barreto-Lynn-Scott elliptic curves are described in [42]. 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

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

The parameters are listed in Table 6.

**Table 6.** Parameters of used Barreto-Lynn-Scott elliptic curves

<b>BLS12-381 [45]</b>	
$k$	12
$r$	5243587517512619047944774050818596583769055250052763782260365 8699938581184513
$q$	4002409555221667393417789825735904156556882819939007885332058 136124031650490837864442687629129015664037894272559787
$\beta$	-1
$b$	4
$\xi$	$1 + u$
$s$	32

## C.2 Barreto-Naehring elliptic curves

Barreto-Naehring elliptic curves are described in [43]. 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 7.

**Table 7.** Parameters of used Barreto-Naehring elliptic curves

<b>BN254 [6]</b>	
$k$	12
$r$	2188824287183927522224640574525727508854836440041603434369820 4186575808495617
$q$	2188824287183927522224640574525727508869631115729782366268903 7894645226208583
$\beta$	-1
$b$	3
$\xi$	$9 + u$
$s$	28
<b>BN183</b>	
$k$	12
$r$	6804759748846355405830582786011032970784946075266449409
$q$	6804759748846355405830582788619626413398422602255236423
$\beta$	-1
$b$	3
$\xi$	$2 + u$
$s$	30
<b>BN124</b>	
$k$	12
$r$	17000133324792832058895897937997463553
$q$	17000133324792832063019019729102503239
$\beta$	-1
$b$	3
$\xi$	$5 + u$
$s$	25

### C.3 Galbraith-McKee-Valença elliptic curves

Galbraith-McKee-Valença elliptic curves are described in [44]. 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 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 8.



**Table 8.** Parameters of used Galbraith-McKee-Valença elliptic curves

<b>GMV181 [6]</b>	
$k$	6
$r$	1552511030102430251236801561344621993261920897571225601
$q$	6210044120409721004947206240885978274523751269793792001
$\beta$	61
$a$	1
$d$	600581931845324488256649384912508268813600056237543024
$\xi$	$u$
$s$	31

<b>GMV97</b>	
$k$	6
$r$	141455844224742490147094691841
$q$	565823376898968604518330826753
$\beta$	5
$a$	5
$d$	482996825047815773983380486779
$\xi$	$u$
$s$	15

<b>GMV58</b>	
$k$	6
$r$	211006452744585217
$q$	844025809322115073
$\beta$	10
$a$	5
$d$	579073710274753001
$\xi$	$u$
$s$	19