The inspection model for zero-knowledge proofs and efficient Zerocash with secp256k1 keys

Huachuang Sun, Haifeng Sun, Kevin Singh,
Akhil Sai Peddireddy, Harshad Patil,
Jianwei Liu, Weikeng Chen *

Discreet Labs

Abstract

Proving discrete log equality for groups of the same order is addressed by Chaum and Pedersen's seminal work [CP92]. However, there has not been a lot of work in proving discrete log equality for groups of different orders.

This paper presents an efficient solution, which leverages a technique we call *delegated Schnorr*. The discovery of this technique is guided by a design methodology that we call *the inspection model*, and we find it useful for protocol designs.

We show two applications of this technique on the Findora blockchain:

- *Maxwell-Zerocash switching:* There are two privacy-preserving transfer protocols on the Findora blockchain, one follows the Maxwell construction and uses Pedersen commitments over Ristretto, one follows the Zerocash construction and uses Rescue over BLS12-381. We present an efficient protocol to convert assets between these two constructions while preserving privacy.
- *Zerocash with secp256k1 keys:* Bitcoin, Ethereum, and many other chains do signatures on `secp256k1`. There is a strong need for ZK applications to not depend on special curves like Jubjub, but be compatible with `secp256k1`. Due to FFT unfriendliness of `secp256k1`, many proof systems (e.g., Groth16, Plonk, FRI) are infeasible. We present a solution using Bulletproofs over curve `secq256k1` ("q") and delegated Schnorr which connects Bulletproofs to TurboPlonk over BLS12-381.

We conclude the paper with *(im)possibility* results about Zerocash with only access to a deterministic ECDSA signing oracle, which is the case when working with MetaMask. This result shows the limitations of the techniques in this paper.

This paper is under a bug bounty program by a grant from the Findora Foundation.

*Authors listed in reverse alphabetical order. Authors can be reached at crypto@findora.org.

Contents

1 Introduction **3**
1.1 Imagining an inspector between prover and verifier . . . 5
1.2 Pushing nonnative group simulation outside NIZK . . . 9
1.3 Application: Maxwell-Zerocash switching . . . 12
1.4 Application: Zerocash with secp256k1 keys . . . 14
1.5 Rest of the paper . . . 18
1.6 Bug bounty program . . . 19

2 Cryptographic building blocks **19**

3 The inspection model for zero-knowledge proofs **20**
3.1 Design rationale . . . 20
3.2 Optimization used in production . . . 22
3.3 Informal security proof . . . 23
3.4 Alternative construction . . . 24

4 Delegated Schnorr protocol **25**
4.1 Batching . . . 25
4.2 Informal security proof . . . 28

5 Implementation and evaluation **29**
5.1 Implementation . . . 29
5.2 Evaluation setup . . . 30
5.3 Evaluation on Maxwell-Zerocash switching . . . 30
5.4 Evaluation on Zerocash with secp256k1 keys . . . 32

6 Open problems **34**

Acknowledgment **35**

A A binary-testing-friendly variant of TurboPlonk **35**
A.1 Indexer . . . 36
A.2 Prover . . . 38
A.3 Verifier . . . 42

B (Im)possibility of fully MetaMask-compatible Zerocash **44**

References **47**

1 Introduction

We start by formalizing the problem of proving discrete log equity for groups of different orders. Consider two prime-order cyclic groups $\mathbb{G}_1$ and $\mathbb{G}_2$ where DLOG is hard, as follows.

- $\mathbb{G}_1$ has prime order p, with two generators G_1 and H_1 sampled from $\mathbb{G}_1$
- $\mathbb{G}_2$ has prime order q, with two generators G_2 and H_2 sampled from $\mathbb{G}_2$

where no polynomial-size adversary can compute $\log_{G_1} H_1$ or $\log_{G_2} H_2$. This is achieved by using a hash function modeled as a random oracle.

Problem. Consider two group elements P and Q. Prover $\mathbf{P}$ knows x_1, x_2, r_1, r_2 such that:

$$P = x_1 G_1 + r_1 H_1 \in \mathbb{G}_1 \text{ where } x_1, r_1 \in \{0, 1, ..., p-1\}$$
$$Q = x_2 G_2 + r_2 H_2 \in \mathbb{G}_2 \text{ where } x_2, r_2 \in \{0, 1, ..., q-1\}$$

Prover $\mathbf{P}$ wants to show to verifier $\mathbf{V}$ that it knows x_1, r_1, x_2, r_2 and that $x_1 = x_2$, without revealing x_1, x_2, r_1, r_2. This is a zero-knowledge proof of knowledge with the following instance $\mathbb{x}$ and witness $\mathbb{w}$.

- Instance $\mathbb{x}$: the group elements P, Q
- Witness $\mathbb{w}$: the scalars x_1, r_1, x_2, r_2

Challenge of using general-purpose zk-SNARK. We can solve this problem using zk-SNARK, such as Groth16 [Gro16], Bulletproofs [BBBPWM18], Stark [BSBHR18], and Plonk [GWC19]. However, it is not easy to find an efficient implementation, for the following reasons:

- **Interoperability with the rest of the system:** Picking a curve for $\mathbb{G}_1$ or $\mathbb{G}_2$ may affect how the zk-SNARK integrates with the rest of the system. Consider a system that implements Zerocash using TurboPlonk over BLS12-381. This is a standard choice because TurboPlonk has a short proof, and the universal setup parameters for BLS12-381 already exist. However, BLS12-381 likely does not favor either of these two groups. Using another curve is challenging, as another setup ceremony may be needed. More generally speaking, there are situations where the overall system cannot afford to cater for a subprotocol's preference of the curve.
- **Embedding curves are large:** Pairing-based zk-SNARKs cover almost all schemes with a proof of size less than ≈ 50 kilobytes except Bulletproofs [BBBPWM18]. Assume that $\mathbb{G}_1$ is a group of points on an elliptic curve. If we do not know a priori another curve that embeds this curve, we need to find such a curve, often by using the Cocks-Pinch algorithm [FST10]. However, curves found in this way are inefficient due to a large ρ-value.
- **Restrictions over FFT-unfriendly fields:** Many zk-SNARK schemes require the field to be FFT-friendly: Groth16 [Gro16], Plonk [GWC19], Stark [BSBHR18], and the like. For a field that is not FFT-friendly, our remaining options include inner-product arguments (Bulletproofs [BBBPWM18] and Spartan [Set20]), linear codes (Brakedown [GLSTW21] and Orion [XZS22]), tensor-product arguments (Gemini [BCHO22]), layered circuits [GKR08; CMT12] (Libra [XZZPS19]), and the like. These schemes are not without disadvantages, but in this case we have to use one of them.

- **Overhead of nonnative group:** Consider the case where $\mathbb{G}_1$ and $\mathbb{G}_2$ are both elliptic curves. Unless $\mathbb{G}_1$ and $\mathbb{G}_2$ have the same coordinate field and the zk-SNARK is working on this field, it is unavoidable to simulate nonnative group operations in zk-SNARK. We did an experiment on the `arkworks-rs` library [ark] simulating MNT4's scalar field over BLS12-381's scalar field. The results indicate a slowdown of $1000\times$ due to being nonnative.

Since general-purpose proof systems seem to be infeasible, we may turn to zero-knowledge protocols tailored for this problem. The most natural solution is to adapt the Chaum-Pedersen protocol, which can prove discrete log equality of groups of the same order.

Challenge of adapting the Chaum-Pedersen protocol. The Chaum-Pedersen protocol [CP92] is an interactive protocol to the problem above when $p = q$. It consists of three messages.

- $\mathbf{P}$ samples $a, b, c \in \mathbb{Z}_p^+$, computes $C_1 := aG_1 + bH_1$, $C_2 := aG_2 + cH_2$, and sends C_1, C_2 to $\mathbf{V}$.
- $\mathbf{V}$ samples challenge $\beta \in \mathbb{Z}_p^+$ and sends β to $\mathbf{P}$.
- $\mathbf{P}$ computes $s_1 := \beta x_1 + a$, $s_2 := \beta r_1 + b$, $s_3 := \beta r_2 + c$ and sends s_1, s_2, s_3 to $\mathbf{V}$.

In the decision round, $\mathbf{V}$ accepts the proof when:

$$s_1 G_1 + s_2 H_1 = \beta P + C_1, \quad s_1 G_2 + s_3 H_2 = \beta Q + C_2$$

This protocol is public-coin and can be made non-interactive through Fiat-Shamir transform [FS86].

However, the protocol does not work when $p \neq q$. Assume that we keep $\mathbf{V}$'s computation above unmodified with the only change that $p \neq q$. A malicious prover $\widetilde{\mathbf{P}}$ can convince $\mathbf{V}$ that $x_1 = x_2$ even though $x_1 \neq x_2$ because $\widetilde{\mathbf{P}}$ only needs to find s_1 that satisfies the following conditions, through the Chinese Remainder Theorem (CRT).

$$s_1 \equiv \beta x_1 + a \pmod{p} \quad s_1 \equiv \beta x_2 + a \pmod{q}$$

The solution s_1 must not be too large (it has an explicit upper bound $(\beta + 1) \cdot \min(p, q)$ and a lower bound $\max(p, q)$, following a certain distribution). $\widetilde{\mathbf{P}}$ in the non-interactive setting can try different a, b, c until such an appropriate s_1 is found.

Another problem is that we did not check $x_1, x_2 \in \{0, 1, ..., \min(p, q) - 1\}$. This missing check can lead to security issues. For example, assume $q > p$, the prover can easily prove the equality for $x_1 = x_2 = p + 1$ though x_1 is out of range.

We could, however, have $\mathbf{P}$ prove the knowledge of x_1, r_1 and x_2, r_2 through two separate invocations of the Schnorr protocol, but this is not meaningful. Doing so only proves that $\mathbf{P}$ has the knowledge, but it does not mean that in this "knowledge" $x_1 = x_2$ holds. This leads to an interesting question that inspires the solution presented in this paper:

Can we, in a proof of knowledge protocol, inspect what the knowledge is?

We show that this is possible, and indeed, the idea of *"inspection"* may help us design protocols.

1.1 Imagining an inspector between prover and verifier

A standard zero-knowledge proof system has two entities: a prover $\mathbf{P}$ and a verifier $\mathbf{V}$. We add a third entity:[1] an inspector $\mathbf{i}$, which stands between the prover and the verifier, as shown in Figure 1.

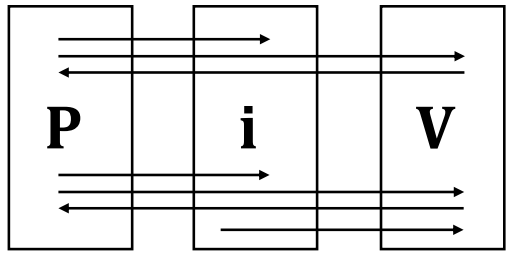

Figure 1: The inspection model: a prover $\mathbf{P}$, an inspector $\mathbf{i}$, and a verifier $\mathbf{V}$.

Informally, the inspector $\mathbf{i}$ does three things:

- read public messages between $\mathbf{P}$ and $\mathbf{V}$
- read private messages sent by $\mathbf{P}$, which $\mathbf{P}$ does not want to share with $\mathbf{V}$
- in the decision phase, send a signal `VETO` to $\mathbf{V}$ if $\mathbf{i}$ wants to veto this proof, or $\perp$ otherwise

In this model, both $\mathbf{P}$ and $\mathbf{V}$ trust the inspector $\mathbf{i}$. Prover $\mathbf{P}$ is willing to share to $\mathbf{i}$ information that $\mathbf{P}$ will not share with $\mathbf{V}$. Verifier $\mathbf{V}$ trusts the veto signal from $\mathbf{i}$: it will reject the proof if $\mathbf{i}$ vetoes. Note that if $\mathbf{i}$ does not veto, $\mathbf{V}$ does not necessarily accept the proof—$\mathbf{V}$ may accept or reject.

Below we compare the inspection model with the standard interactive zero-knowledge proof setting. Message passing in each round is more complicated. However, the definitions of completeness, soundness, zero-knowledge, and knowledge soundness all remain unchanged.

Differences in a round. Originally, in each round, $\mathbf{P}$ sends a message to $\mathbf{V}$, and $\mathbf{V}$ then sends a message back to $\mathbf{P}$. That is, in the i-th round,

- $\mathbf{P}$ sends a message $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{V}}$ to $\mathbf{V}$
- $\mathbf{V}$ sends a message $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P}}$ to $\mathbf{P}$

In the inspection model, the inspector $\mathbf{i}$ will be inspecting the transcript, and in addition, $\mathbf{P}$ shares additional information with $\mathbf{i}$. That is, in the i-th round,

- $\mathbf{P}$ sends a message $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ to $\mathbf{i}$
- $\mathbf{P}$ sends a message $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ to $\mathbf{i}$ and $\mathbf{V}$
- $\mathbf{V}$ sends a message $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ to $\mathbf{P}$ and $\mathbf{i}$

In addition, in the decision round, $\mathbf{i}$ may send the `VETO` signal to $\mathbf{V}$.

- $\mathbf{i}$ sends the `VETO` signal or $\perp$ to $\mathbf{V}$

[1] We use lowercase `\mathbf{i}` for the inspector $\mathbf{i}$ to avoid confusion with the indexer in preprocessing SNARKs, which is often represented in the literature as uppercase `\mathbf{I}`, that is, $\mathbf{I}$.

The definitions of transcript are also more tricky now. To be compatible with existing terminology, we say that the transcript includes all the messages between $\mathbf{P}$ and $\mathbf{V}$, and an *extended* transcript includes messages between $\mathbf{P}$, $\mathbf{i}$, and $\mathbf{V}$. As a result, an extended version of Fiat-Shamir transform [FS86] will be needed, which is a compiler that makes the protocol non-interactive.

Properties remain unchanged. We reiterate the definitions of properties of an interactive zero-knowledge proof system to show that they are unchanged. We do not specify if a property must be perfect or computational or the complexity of an algorithm, as this may vary between protocols, and is independent of the inspection model here.

- **Completeness:** For an instance-witness pair in the relation, $\mathbf{P}$ can convince $\mathbf{V}$ to accept the proof with some probability.[2]
- **Soundness:** For an instance-witness pair not in the relation, a malicious prover $\widetilde{\mathbf{P}}$ can convince $\mathbf{V}$ to accept the proof except with some probability.
- **Zero-knowledge:** For an instance-witness pair in the relation, the view of a malicious verifier $\widetilde{\mathbf{V}}$ can be simulated by a simulator with some access to $\widetilde{\mathbf{V}}$ except with some probability.
- **Knowledge soundness:** An extractor with some sort of access to the malicious prover $\widetilde{\mathbf{P}}$ can find a witness in the relation with a probability higher than the probability that $\widetilde{\mathbf{P}}$ convinces $\mathbf{V}$. This implies soundness.

Compiler to NIZK. We will present a modified version of the Fiat-Shamir transform that can convert an interactive proof in the inspection model that is (1) public-coin, (2) perfect complete, (3) computational knowledge sound, and (4) computational honest-verifier zero-knowledge to an NIZK, through a black-box use of an NIZK scheme.

This means that although in the real world it is unlikely to find an inspector $\mathbf{i}$, the inspection model is still useful for writing and designing an interactive proof protocol that will later be transformed into an NIZK. This is in essence similar to the common approach to building zk-SNARK from interactive oracle proof (IOP) and polynomial commitments (PC), where the polynomials are modeled as oracles, which do not exist in the real world, and are later instantiated with PC.

Application: inspecting the knowledge in the proof of knowledge in the Schnorr protocol. We present an example to show the usefulness of the inspection model in describing the problem. The Schnorr protocol for a Pedersen commitment $P = xG + rH$ produces a proof of knowledge, stating that $\mathbf{P}$ knows x and r, but we cannot easily place additional constraints over x, for example, $0 \leq x < 2^{64}$, although it feels like just "a small addition" to the problem, it seems difficult to do it with the Schnorr protocol.

Let us now revisit the Schnorr protocol for proof of knowledge of x and r here, as Figure 2 shows. Then, we will discuss an extended version of the protocol, in the inspection model, where the knowledge is being inspected by $\mathbf{i}$.

1. $\mathbf{P}$ samples $a, b \leftarrow\$ \mathbb{F}$, computes $C := aG + bH$, and sends it to $\mathbf{V}$
2. $\mathbf{V}$, which is an honest verifier, samples $\beta \leftarrow\$ \mathbb{F}$ and sends β to $\mathbf{P}$
3. $\mathbf{P}$ computes $s_1 := \beta x + a$ and $s_2 := \beta r + b$ and sends s_1 and s_2 to $\mathbf{V}$

[2] This implies that in such a situation, the inspector $\mathbf{i}$ will send $\perp$ to $\mathbf{V}$ in the decision round, rather than vetoing.

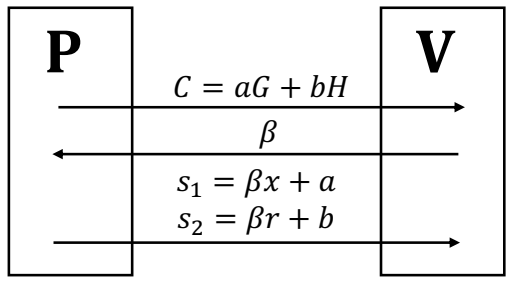

Figure 2: The Schnorr protocol for proof of knowledge.

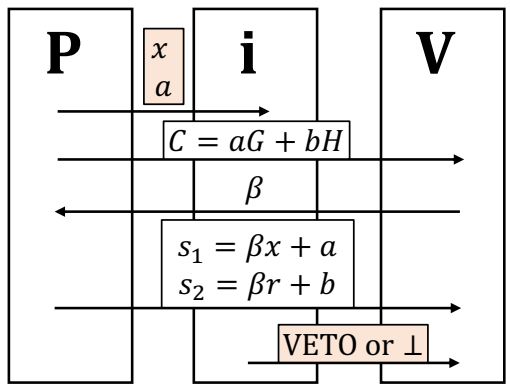

Figure 3: A modified version of the Schnorr protocol in the inspection model.

4. **V** decides by checking if $s_1 G + s_2 H \stackrel{?}{=} \beta P + C$

Now we discuss how an inspector **i** can help enforce $0 \leq x < 2^{64}$ for the "knowledge" x, which is also equipped with only "a small addition", as shown in Figure 3.

1. **P** samples $a, b \leftarrow\$ \mathbb{F}$, computes $C := aG + bH$, sends x and a to **i**, and sends C to **i** and **V**
2. **V** samples $\beta \leftarrow\$ \mathbb{F}$ and sends β to **P** and **i**
3. **P** computes $s_1 := \beta x + a$ and $s_2 := \beta r + b$ and sends s_1 and s_2 to **i** and **V**
4. **i** decides whether or not to vote by checking if $0 \leq x < 2^{64}$ and $s_1 \stackrel{?}{=} \beta x + a$
5. **V** decides by checking if **i** did not veto and checking if $s_1 G + s_2 H \stackrel{?}{=} \beta P + C$

We now explain the high-level idea of the security proof. Assume that **P** sends x' and a' in the first message. Here, x' does not necessarily match the value x committed in the Pedersen commitment P, and a' does not necessarily match the value a committed in the Pedersen commitment C in **P**'s next message. If **V** accepts the proof, then it means that $s_1 = \beta x + a$, and the inspector **i** did not veto, which implies the following equation:

$$\beta x + a = \beta x' + a'$$

If $x = x'$, then $a = a'$. Otherwise, **P** needs to choose a' in the following way:

$$a' = \beta(x - x') + a$$

But β is sampled later by $\mathbf{V}$, and therefore the probability that $\mathbf{P}$ can provide such an a' is negligible for sufficiently large $\mathbb{F}$. In conclusion, we have $x' = x$, and in addition, $0 \leq x < 2^{64}$.

The remaining question is how to instantiate the inspector $\mathbf{i}$ in the real world. The high-level idea is to run the inspector $\mathbf{i}$ in NIZK and use a simulatable commitment scheme with extractability.

Application: discrete log equality for groups of different orders. Note that the protocol above immediately implies a solution to the problem at the beginning: one can run the protocol above twice, one for the element of each group, and ask inspector $\mathbf{i}$ to check the discrete log equality. In the rest of the paper, we focus on the modified version of the Schnorr protocol, which is more general when integrated with other protocols and is sufficient for our two applications.

Sketch of the compiler. We provide the necessary background about the compiler, which performs a modified Fiat-Shamir transform and works as follows. As we mentioned, it requires the interactive proof to be public-coin, with perfect completeness, computational knowledge soundness, and computational honest-verifier zero-knowledge. As follows and in the rest of the paper, we use $\mathbb{F}_G$ to describe the scalar field of $\mathbb{G}$ and $\mathbb{F}_N$ to describe the field where the NIZK is situated.

• **Proof generation:**

1. Instantiate a transcript T for the Fiat-Shamir transform
2. Put the transcript header and the instance $\mathbb{x}$ into the transcript
3. Process each round while running the prover $\mathbf{P}$. For the i-th round,
   - "$\mathbf{P}$ *sends a message* $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ *to* $\mathbf{i}$" will be processed as follows:
     (a) Sample a random blinding factor $r_i \in \mathbb{F}_N$
     (b) Compute a commitment $c_i := H(\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}; r_i)$
     (c) Output c_i as part of the proof
     (d) Put c_i into the transcript
   - "$\mathbf{P}$ *sends a message* $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ *to* $\mathbf{i}$ *and* $\mathbf{V}$" will be processed as follows:
     (a) Output $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ as part of the proof
     (b) Put $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ into the transcript
   - "$\mathbf{V}$ *sends a message* $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ *to* $\mathbf{P}$ *and* $\mathbf{i}$" will be processed as follows:
     (a) Do nothing, since $\mathbf{V}$ is public-coin, $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ can be computed from the transcript, through the Fiat-Shamir transform
4. For the decision round:
   - Generate an NIZK π with the following instance $\mathbb{x}$, witness $\mathbb{w}$, and statement Π:
     – Instance $\mathbb{x}$: all the $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$, all the $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$, and all the c_i
     – Witness $\mathbb{w}$: all the $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$, and all the r_i
     – Statement Π:
       ∗ Check for all i: $c_i \stackrel{?}{=} H(\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}; r_i)$
       ∗ Check that $\mathbf{i}([\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}, \mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}, \mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}]_i) \stackrel{?}{=} \perp$
   - Output π as part of the proof

• **Proof verification:**

1. Instantiate a transcript T for the Fiat-Shamir transform
2. Put the transcript header and the instance $\mathbb{x}$ into the transcript
3. Process each round while running the prover $\mathbf{V}$. For the i-th round,
   - "$\mathbf{P}$ *sends a message* $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ *to* $\mathbf{i}$" will be processed as follows:
     - Put c_i into the transcript
   - "$\mathbf{P}$ *sends a message* $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ *to* $\mathbf{i}$ *and* $\mathbf{V}$" will be processed as follows:
     - Put $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ into the transcript
   - "$\mathbf{V}$ *sends a message* $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ *to* $\mathbf{P}$ *and* $\mathbf{i}$" will be processed as follows:
     - Do nothing, since $\mathbf{V}$ is public-coin, $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ can be computed from the transcript, through the Fiat-Shamir transform
4. For the decision round:
   - Verify the NIZK π using the instance—all the $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$, all the $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$, and all the c_i—which come from the proof or from the transcript via Fiat-Shamir transform
   - Check that $\mathbf{V}([\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}, \mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}]_i) \stackrel{?}{=} \texttt{ACCEPT}$

We will discuss the design rationale, optimization, and security in Section 3. Now we focus on one application—reading the message in a Pedersen commitment—where the group that Pedersen commitment is defined over may not be "native" to the NIZK and requires field simulation, such as a zk-SNARK defined over BLS12-381's scalar field that wishes to read a Pedersen commitment on `curve25519` or `secq256k1`.

1.2 Pushing nonnative group simulation outside NIZK

We will now describe the delegated Schnorr protocol, which enables an NIZK to read the message in a Pedersen commitment, without performing nonnative group simulation inside the NIZK. We first start with a strawman solution that does not use this protocol. As follows, we use groups on elliptic curves as an example.

Strawman: doing nonnative scalar-point multiplication in NIZK. A straightforward solution to show that $P = xG + rH$ is to, given G, H, x, r, compute the scalar-point multiplication xG and rH and then add these two points together inside NIZK. Such computation is not expensive if the NIZK works well with the coordinate field of this curve. For example, for pairing-based NIZK using BLS12-381, the Jubjub curve [Zcab] and the Bandersnatch curve [MSZ21] are two such curves. However, what if this is not the case?

	# Constraints (constraint-optimized)	# Nonzeros in R1CS matrices (weight-optimized)
BLS12-381 simulating `secp256k1/curve25519`	673	3541

Table 1: Overhead of field simulation in the `arkworks-rs` library.

If this NIZK over BLS12-381 needs to do scalar-point multiplication on a curve with a coordinate field that is not supported natively, it is still possible but with a high cost. To simulate computation over another field $\mathbb{F}' \neq \mathbb{F}$ from a field $\mathbb{F}$, the field element needs to be represented by a number of "limbs", which are field elements in $\mathbb{F}$. When one wants to simulate the modular multiplication of two elements $\mathbb{F}'$ [KPS18; OWWB20], it becomes rather complicated. We show in Table 1 the number of constraints (in a constraint-optimized construction) or the number of nonzeros in R1CS matrices (in a weight-optimized construction) for BLS12-381 to simulate a 256-bit prime field such as `secp256k1` and `curve25519`. The overhead of many proof systems is closely related to the number of nonzeros, and this means that field simulation incurs an overhead of about $1000\times$.

Delegation: pushing nonnative group simulation outside NIZK. As follows, we describe the delegated Schnorr protocol, which we use to push the nonnative group simulation outside NIZK, so inside there are only a few nonnative field simulation operations. We obtain this protocol by running the compiler over the modified Schnorr protocol in the inspection model (in Figure 3), but without the range check $0 \leq 0 < 2^{64}$. Although the compiler provides all messages to the inspector $\mathbf{i}$ in the NIZK, inspector $\mathbf{i}$ may only use some of them. We use gray to indicate these unused messages. These messages do not need to be provided to NIZK in production.

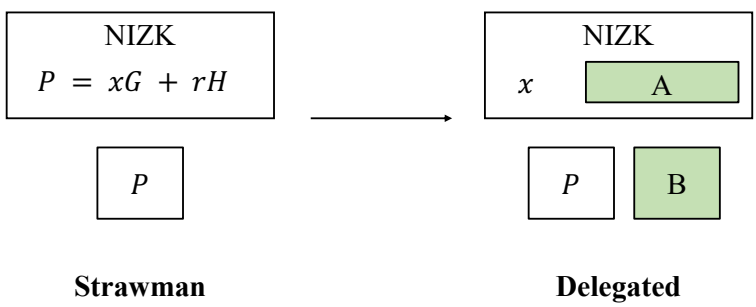

Figure 4: An illustration that NIZK delegates the scalar multiplication outside.

We now describe how prover $\mathbf{P}$ generates the overall proof, which consists of the NIZK with new statements (block **A**) and the auxiliary proof information (block **B**), followed by how verifier $\mathbf{V}$ verifies the overall proof.

• **Proof generation:**

1. Sample $a, b \leftarrow\$ \mathbb{F}_G$, compute $C := aG + bH$, sample $r_1 \leftarrow\$ \mathbb{F}_N$, compute $c_1 := H(x, a; r_1)$, and put (c_1, C) into the transcript
2. Compute the challenge $\beta \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
3. Compute $s_1 := \beta x + a$ and $s_2 := \beta r + b$ and put them into the transcript
4. Set block B to be (c_1, C, s_1, s_2)
5. Append the following statement (block A) into NIZK:
   - Additional instance $\mathbb{x}^+$: c_1, C, β, s_1, s_2
   - Additional witness $\mathbb{w}^+$: x, a, r_1
   - Additional statement Π^+:

- Check $c_1 = H(x, a; r_1)$
- Check $s_1 = \beta x + a$

6. Proceed with the rest of the protocol

• **Proof verification:**

1. Parse block B as (c_1, C, s_1, s_2)
2. Compute the challenge $\beta \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
3. Check that $\beta P + C = s_1 G + s_2 H$
4. Check the NIZK proof with the additional instance $\mathbb{x}^+ := (c_1, \beta, s_1)$

Comparison. We now compare the number of operations done in the strawman solution and the delegated solution. Let p be the order of the group $\mathbb{G}$ and let n be the number of bits in p. We present the result in Table 2, focusing on the dominating cost. As one can see, in the delegated solution, the NIZK only needs to perform one nonnative multiplication in $\mathbb{F}$, while in the strawman solution, the NIZK needs to compute xG and rH (each has $4(n-1)$ nonnative muls) and then compute $xG + rH$ (it has 4 nonnative mul). In practice, for a secure elliptic curve, n is at least around 256 bits, which means that the strawman needs to perform ≈ 2000 nonnative muls. The delegated solution, however, comes with a small overhead, in that the three scalar-point multiplication operations are done outside NIZK.

	NIZK	**non-NIZK**
Strawman	$2 \cdot 4(n-1) + 4$ nonnative muls	—
Delegated	1 nonnative mul	3 scalar-point muls

Table 2: Comparison of the number of operations incurred by these two solutions.

We also compare the proof size in Table 3, but the result is not conclusive, as we do not know the relationship between the size of the NIZK and the statement that it is proving.

	Proof size
Strawman	The size of the NIZK for the statement with $P = xG + rH$
Delegated	The size of the NIZK for the statement in block A + 3 elements in $\mathbb{F}$ + 1 element in $\mathbb{G}$

Table 3: Comparison of the proof size.

Finally, we want to provide an advance notice that Table 2 and Table 3 did not show the case when more than one point is read. Indeed, if we read K points,

- The strawman solution performs $K \cdot (2 \cdot 4(n-1) + 4)$ nonnative muls in NIZK, and the proof size is the size of the NIZK proof for such a statement.
- The delegated solution performs 1 nonnative mul in NIZK (regardless of K) and $(2K+1)$ scalar-point muls (at the cost of field operations for batching), and the proof size is the size of the NIZK proof for such a statement, plus $(2+K)$ elements in $\mathbb{F}$ (with batching) and K elements in $\mathbb{G}$

We will discuss batching in Section 4. To conclude, if the size of NIZK in the strawman solution grows slower than that in the delegated solution, then the delegated Schnorr protocol can be viewed as that it trades off space for proof generation time.

1.3 Application: Maxwell-Zerocash switching

We consider two privacy-preserving transfer protocols, one follows the Maxwell construction, one follows the Zerocash construction. These two constructions represent private assets in different ways, and therefore, it is desired to be able to directly convert one to another, without revealing the information inside.[3] We first provide the necessary background of these two constructions.

Background: Maxwell construction. In 2015, a renowned blockchain developer Gregory Maxwell presented a construction of confidential transactions (CT) [Ele; PBFMW18] for a UTXO chain. An asset can be viewed as consisting of three parts of the information: the owner's address, the amount, and the asset type. The Maxwell construction hides the amount and the asset type, but it does not hide the owner's address. To do so, each asset is now represented by an address and two Pedersen commitments, one for the amount, and one for the asset type, as shown in Table 4.

Asset attribute	Representation
The owner's address	Same as in the transparent blockchain
Amount (denoted by x)	Pedersen commitment $P = xG + \gamma_1 H$ (γ_1 is a randomizer)
Asset type (denoted by y)	Pedersen commitment $Q = yG + \gamma_2 H$ (γ_2 is a randomizer)

Table 4: Representation of an asset in the Maxwell construction.

In the Maxwell construction, to prove that an asset transfer is valid, one needs to show that the inputs and outputs of this asset transfer are equal for each of the asset types. This is done, in the Findora blockchain, through the use of sigma proofs and/or Bulletproofs, defined over the Ristretto group in `curve25519`. The choice of Ristretto is based on the fact that pairing is not needed and that this curve is known to be performant. The concrete instantiation is described in the Bulletproofs paper [BBBPWM18].

Background: Zerocash construction. Improved over a prior protocol Zerocoin [MGGR13], the Zerocash construction [BS+14], firstly proposed by Ben-Sasson, Chiesa, Garman, Green, Miers, Tromer, and Virza and improved subsequently through the efforts of the Zcash foundation [Zfn] and Electric Coin Company (ECC) [Ecc], is another privacy-preserving transfer protocol over a UTXO chain. The Zerocash construction offers stronger privacy guarantees, as it also hides the owner's address, which is the best we can achieve.

[3] Some readers may wonder why the Findora blockchain keeps and actively supports both the Maxwell construction and the Zerocash construction, even though the Zerocash construction offers stronger privacy guarantees. This decision is not just for backward compatibility—we see a lot of advantages of the Maxwell construction over the Zerocash construction. For example, the Maxwell construction represents assets as Pedersen commitments, which are easy to integrate with other systems through sigma protocols, and it takes much less on-chain resources.

To do that, in the Zerocash construction, an asset is an (unspent) cryptographic commitment, which commits the owner's address, the amount, and the asset type, in a global Merkle tree. Now, to spend an asset, the owner computes and publishes a nullifier that is unique for this commitment. This can prevent the owner from spending the commitment again as the network can reject transactions with duplicated nullifiers. We summarize the representation of an asset in the Zerocash construction, the commitment and its nullifier, in Table 5.

	Commitment	**Nullifier**
Instantiation	A hash commitment of • the owner's address • the amount • the asset type with randomizer r	A hash commitment of • the owner's address • the amount • the asset type • the owner's private key with the same randomizer r

Table 5: Representation of an asset in the Zerocash construction.

This is done, in the Findora blockchain, through the use of a binary-testing-friendly variant of TurboPlonk (described in Appendix A), over the BLS12-381 curve, where the hash commitment is instantiated using the Rescue hash function [AABSDS20]. This choice is based on a number of reasons. We choose pairing-friendly proof systems because we want short proofs and we cannot afford Bulletproofs in this case due to its high proving and verifying costs for this kind of statement. BLS12-381 is a pairing-friendly curve whose universal setup parameters exist (from Zcash's Powers of Tau parameter generation [Par]) and whose Hamming weight of some pairing parameters is low. TurboPlonk is a pairing-based zero-knowledge proof system that enables customized gates, which, according to the sparsity theory, can trade off FFT for MSM and improve the performance severalfold. Rescue is a SNARK-friendly hash function that can be efficiently implemented using customized gates in TurboPlonk.

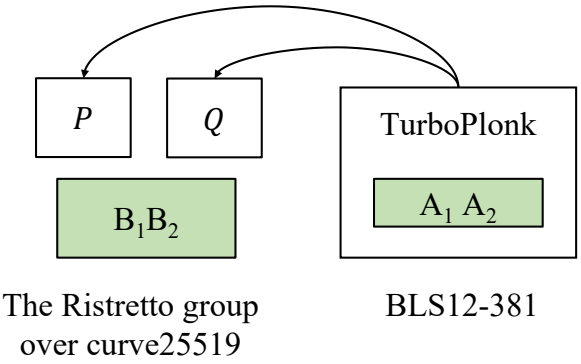

Figure 5: Illustration of the Maxwell-to-Zerocash switching, in which an NIZK over BLS12-381 reads two Pedersen commitments on the Ristretto group.

Maxwell-to-Zerocash. We now briefly describe how we switch a Maxwell asset to a Zerocash

asset, which is done through a delegated Schnorr proof working together with TurboPlonk over BLS12-381. This is the case when the NIZK (here, TurboPlonk) needs to read two Pedersen commitments on Ristretto. The protocol is illustrated in Figure 5.

We now describe the protocol for completeness.

• **Proof generation:**

1. Sample $a, b, c, d \leftarrow\$ \mathbb{F}_G$, compute $C_1 := aG + bH$ and $C_2 := cG + dH$, sample $r_1 \leftarrow\$ \mathbb{F}_N$, compute $c_1 := H(x, a, y, c; r_1)$, and put (c_1, C_1, C_2) into the transcript
2. Compute the challenge $\beta \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
3. Compute $s_1 := \beta x + a$, $s_3 := \beta y + c$ and put them into the transcript
4. Compute the challenge $\lambda \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
5. Compute $s_{24} := (\beta\gamma_1 + b) + \lambda(\beta\gamma_2 + d)$ and put it into the transcript
6. Set block $\mathrm{B}_1\mathrm{B}_2$ to be $(c_1, C_1, C_2, s_1, s_3, s_{24})$
7. Append the following statement (block $\mathrm{A}_1\mathrm{A}_2$) into NIZK:
   - Additional instance $\mathbb{x}^+$: c_1, C_1, C_2, β, λ, s_1, s_3, s_{24}, as well as instance for precomputation, $\beta\lambda$ and $s_1 + \lambda s_3$, which are computed outside and provided as instance to NIZK
   - Additional witness $\mathbb{w}^+$: x, a, y, c, r_1
   - Additional statement Π^+:
     - Check $c_1 = H(x, a, y, c; r_1)$
     - Check $(s_1 + \lambda s_3) = \beta x + a + (\beta\lambda)y + \lambda c$
8. Sample $r \leftarrow\$ \mathbb{F}_N$ for the randomizer in the Rescue hash commitment
9. Compute the Rescue hash commitment that represents the Zerocash asset: $h := H(\mathrm{PK}, x, y; r)$ where PK is the address of the owner of the new Zerocash asset
10. Proceed with the rest of the protocol

• **Proof verification:**

1. Parse block B as $(c_1, C_1, C_2, s_1, s_3, s_{24})$
2. Compute the challenge $\beta, \lambda \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
3. Check that $\beta P + C_1 + \lambda(\beta Q + C_2) = (s_1 + \lambda s_3)G + s_{24}H$
4. Check the NIZK proof with the additional instance $\mathbb{x}^+ := (c_1, \beta, \lambda, \beta\lambda, s_1 + \lambda s_3)$ where $\beta\lambda$ and $s_1 + \lambda s_3$ are computed outside

Zerocash-to-Maxwell. The other direction of the protocol is straightforward. First, we spend the Zerocash asset in the Zerocash construction, which outputs a nullifier of this Zerocash asset. Then, instead of generating a new Zerocash asset, it reads the new Maxwell asset outside, through the delegated Schnorr protocol, and checks that it has the same amount and asset type.

1.4 Application: Zerocash with secp256k1 keys

We now present an application of the delegated Schnorr protocol, which is used in Zerocash to support signing via the secp256k1 keys. We concretely instantiate and implement this application

with two building blocks: (1) Bulletproofs over `secq256k1` ("q"), which can efficiently verify a scalar multiplication relation of points in `secp256k1` ("p") and (2) delegated Schnorr protocol that reads the input commitments of this Bulletproofs, in which the NIZK is the binary-testing-friendly variant of TurboPlonk over BLS12-381 (see Appendix A), and the group to read is points over the `secq256k1` curve ("q").[4]

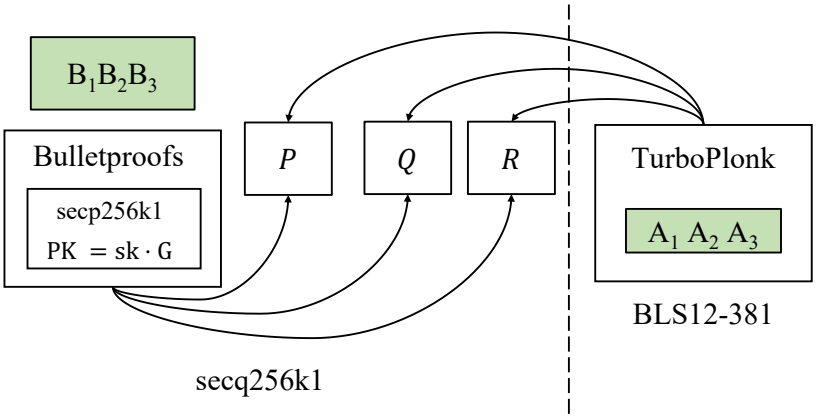

Figure 6: The instantiation of the Zerocash with `secp256k1` keys.

Our design is illustrated in Figure 6. The entire proof is generated in the following manner:

1. Initialize a transcript T, which will be used for the Fiat-Shamir transform throughout the entire proof generation
2. Put the hash h of the transaction body into the transcript T, in which the transaction body includes the nullifiers of spent Zerocash assets, the commitments of new Zerocash assets, as well as other information except for the proof above
3. Let PK $= (x, y)$ be the public key and let sk be the corresponding secret key. Commit $x, y,$ sk using Pedersen commitments with randomizers $\gamma_1, \gamma_2, \gamma_3 \leftarrow\$ F_B$ over `secq256k1`, where F_B is the coordinate field of `secp256k1`, which is the scalar field of `secq256k1`.
   - $P = x \cdot G_{\texttt{secq}} + \gamma_1 \cdot H_{\texttt{secq}}$

[4]Readers may wonder why `secq256k1` rather than another curve is used in our implementation. In fact, we previously found another curve using the complex multiplication method [Jan; BS07; SSKK01], named the Canaan curve, which is $y^2 = x^3 + ax + b \pmod q$ of order r, where:

- $a = 55355509530204640337746971790687835372932334003269362447236185884715359467 49$
- $b = 36647759370566527599092766378540222398030651415577287046115147687263277949759$
- $r =$ `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffefffffc2f`
- $q =$ `0x1000000000000000000000000000000011225471b50b8dc249e5ff726d4163f21`

Nevertheless, we later found `secq256k1` [Poe18], which is almost the same as `secp256k1` except that the order r and the modulus q are swapped. Our comparison shows that Canaan might only have an advantage in hashing-to-curve, which is harder for `secq256k1` given its j-invariant and the lack of a low-degree isogeny map to a curve with nonzero j-invariant, which makes the Wahby-Boneh map [WB19] infeasible. However, we do not need hashing-to-curve for now. We believe that we are not the first human being, also not the last, who attempted to sample their own curve that embeds `secp256k1` and later found out that many years ago people have found `secq256k1`.

- $Q = y \cdot G_{\mathtt{secq}} + \gamma_2 \cdot H_{\mathtt{secq}}$
- $R = \mathsf{sk} \cdot G_{\mathtt{secq}} + \gamma_3 \cdot H_{\mathtt{secq}}$

Note that we can put sk here as it is because the `secp256k1`'s scalar field modulus is smaller than its coordinate field modulus.

4. Generate a Bulletproofs over input commitments P, Q, R using T as the transcript:
   - Check a bit decomposition of sk, denoted by $[b_i]_1^n$; note that here it does not need to be the unique bit decomposition, i.e., the number represented by $[b_i]_1^n$ can be larger than the modulus of the `secp256k1`'s scalar field
   - Check the computation that, on `secp256k1`, $\mathrm{PK}' := \mathrm{sk} \cdot G_{\mathtt{secp}}$ using bit decomposition $[b_i]_1^n$
   - Check that $\mathrm{PK} \stackrel{?}{=} \mathrm{PK}'$

   Note that the Bulletproofs, which continues working on transcript T, serves as a signature of the body hash h; this is a non-black-box use of the non-malleability of Bulletproofs [GOPTT22].
5. Sample $a, b, c, d, e, f \leftarrow\$ F_B$, compute $C_1 := a \cdot G_{\mathtt{secq}} + b \cdot H_{\mathtt{secq}}$, $C_2 := c \cdot G_{\mathtt{secq}} + d \cdot H_{\mathtt{secq}}$, and $C_3 := e \cdot G_{\mathtt{secq}} + f \cdot H_{\mathtt{secq}}$, sample $r_1 \leftarrow\$ \mathbb{F}_N$ where $\mathbb{F}_N$ is the scalar field of BLS12-381, compute $c_1 := H(x, a, y, c, \mathrm{sk}, e; r_1)$, and put (c_1, C_1, C_2, C_3) into the transcript
6. Compute the challenge $\beta \in \mathbb{F}_B$ from the transcript through the Fiat-Shamir transform and compute $s_1 := \beta x + a$, $s_3 := \beta y + c$, and $s_5 := \beta \cdot \mathrm{sk} + e$ and put them into the transcript
7. Compute the challenge $\lambda \in \mathbb{F}_B$ from the transcript through the Fiat-Shamir transform and compute $s_{246} := (\beta\gamma_1 + b) + \lambda(\beta\gamma_2 + d) + \lambda^2(\beta\gamma_3 + f)$ and put it into the transcript and set block $\mathrm{B}_1\mathrm{B}_2\mathrm{B}_3$ to be $(c_1, C_1, C_2, C_3, s_1, s_3, s_5, s_{246})$
8. Append the following statement (block $\mathrm{A}_1\mathrm{A}_2\mathrm{A}_3$) into NIZK:
   - Additional instance $\mathbb{x}^+$: c_1, C_1, C_2, C_3, β, λ, s_1, s_3, s_5, s_{246}, and the instance for precomputation, $\beta\lambda$, λ^2, $\beta\lambda^2$, $s_1 + \lambda s_3 + \lambda^2 s_5$, which are computed outside as instance to NIZK
   - Additional witness $\mathbb{w}^+$: $x, a, y, c, \mathrm{sk}, e, r_1$
   - Additional statement Π^+:
     - Check $c_1 = H(x, a, y, c, \mathrm{sk}, e; r_1)$
     - Check $(s_1 + \lambda s_3 + \lambda^2 s_5) = \beta x + a + (\beta\lambda)y + \lambda c + (\beta\lambda^2)\mathrm{sk} + (\lambda^2)e$
     - Check $0 \leq \mathrm{sk} < |\mathbb{G}_{\mathtt{secp}}|$, where $|\mathbb{G}_{\mathtt{secp}}|$ is the order of the `secp256k1` curve
     - Check that $\mathrm{PK} := (x, y)$ is the public key indicated in the commitments of the Zerocash assets to be spent
     - Check that sk is the secret key used in the nullifiers of these Zerocash assets to be spent
9. Proceed with the rest of the protocol

We want to remind the readers that the checking of the equation $(s_1 + \lambda s_3 + \lambda^2 s_5) = \beta x + a + (\beta\lambda)y + \lambda c + (\beta\lambda^2)\mathrm{sk} + (\lambda^2)e$ is over the scalar field of `secq256k1`. This is the only part that requires field simulation.

The Schnorr-style use of the transcript. Note that non-malleability is necessary here because, otherwise, a malicious user can reuse the Bulletproofs already on chain and pretend to be another user. That is, if without non-malleability guarantees, the protocol above is not checking if the

adversary knows the secret key, it is checking if the adversary *has a proof* of the knowledge of the secret key. What we do here is to push the hash of the transaction body h into the transcript and then Bulletproofs, which is a public-coin protocol that goes through Fiat-Shamir transform, will continue on this transcript. This is similar to the Schnorr signature [Sch89], which is the result of applying the Fiat-Shamir transform to the Schnorr identification protocol. In our case, as a result of such a "Schnorr-style" use of the transcript, now a Bulletproofs of knowledge of the secret key only works for a specific transaction that the prover $\mathbf{P}$ acknowledges.

Importance of Zerocash with secp256k1 keys. We presented the construction first because it flows naturally from previous discussions. Now, we come back and shed some light on the motivation. Since the invention of zk-SNARK, performance has always been a concern. Therefore, researchers and practitioners have been building SNARK-friendly cryptographic primitives, such as the Bowe-Hopwood-Pedersen collision-resistant hash function [Pet18], Rescue [AABSDS20], Poseidon [GKRRS21], Reinforced Concrete [GKLRSW21], and MiMC [AGRRT16; Alb+19]. There is also strong complexity-theoretic evidence that the need to specifically build cryptographic primitives that are SNARK-friendly could be inherent [CL20].

As a result, many zero-knowledge proof systems for blockchain in production replace non-SNARK-friendly primitives with these SNARK-friendly primitives—particularly, the signature scheme. As a result, a lot of these systems have a different address format because the public key of their signature schemes is defined over a curve different from `secp256k1`.

- ZkSync from the Matter Labs uses alternative Baby Jubjub [Zks], a twisted Edwards curve embedded by BN254
- Loopring uses Baby Jubjub [Loo], another twisted Edwards curve embedded by BN254
- Polygon Hermez uses Baby Jubjub [Pol]
- Aztec uses Grumpkin [Azt], which is the curve obtained by swapping the base field and the scalar field of BN254 (since BN254's j-invariant is zero), and it forms a cycle, though not fully pairing-friendly, with BN254
- Starkware uses the STARK curve [Sta], whose coordinate field is a STARK-friendly prime number that is used for the rest of the STARK protocol: $p = 2^{251} + 17 \cdot 2^{192} + 1$

This, however, has caused a lot of user experience issues in production. Today, it is common for a user to use the same public key and address across different EVM-compatible chains, many of which are using the `secp256k1` curve. This includes a lot of the most popular chains—Ethereum, Binance, Avalanche, Polygon, Optimism, Arbitrum, Fantom, Celo, IoTeX, Harmony, and Evmos. Especially, MetaMask currently only supports addresses over the `secp256k1` curve.

Using the same address and public key across different chains has far-reaching benefits in terms of user experience. For example, if user A wants to send tokens to user B, user A only needs to know one address of user B, probably found through the Ethereum Name Service (ENS) [Ens], to be able to send tokens through different chains, and user A is certain that user B can retrieve those tokens, with a small network change on MetaMask, even if user B has never used that specific chain.

The lack of such compatibility is behind the challenges of the adoption of layer-2 zk-Rollup—users of these systems do not have other users that they can send tokens to. It is a common situation that a user of zk-Rollup is unable to find any of his/her friends who is using any zk-Rollup. This is best

illustrated by user feedback in one zk-Rollup's Discord server, saying that such a zk-Rollup that can only do money transfer is "boring".

An example of such "registration-free" compatibility is that anonymous users can send ETH from Tornado Cash to notable individuals [Cryb; Coi] because all it needs is an EVM-compatible address. If zk-Rollup, as well as other layer-1 ZK chains, chooses to have such compatibility, then a user can eventually send tokens to someone who has not registered in that network. This can solve a major, probably the most inherent, obstacle for the adoption of zk-Rollup, as the lack of such compatibility has severe impacts on user experience, and is an unfortunate consequence of the use of SNARK-friendly cryptographic primitives.

The industry is aware of this issue, and we now describe some efforts in this direction.

Related work: existing efforts for doing secp256k1 keys in ZK. The idea of finding a zero-knowledge proof that can efficiently verify ECDSA signatures on `secp256k1` has been there for many years. Indeed, this is one of the main motivations of Bulletproofs [BBBPWM18], which is actually benchmarked over the `secp256k1` curve. To add more detail, one of the authors of Bulletproofs, Andrew Poelstra, is the person who suggested `secq256k1` ("q") [Poe18] in 2018.

The use of the cycle of `secp256k1` ("p") and `secq256k1` ("q") in Zcash has been proposed [Zcaa], however, it is not being considered due to the lack of 2-arity of `secp256k1`. Although 2-arity is not strictly needed for inner-product arguments, the arithmetization of Halo2 will benefit a lot in terms of performance if it can use Lagrange bases, according to the sparsity theory.

Starkware has been researching an extremely new mathematical tool, ECFFT, to bypass the limitation that fast Reed–Solomon interactive oracle proofs of proximity (FRI) [BSCKL21; BSCKL22] require FFT. The main motivation is to help the Cairo programming language [Cai] to avoid non-native fields for verifying ECDSA signatures over `secp256k1`.

To sum up, the need to find SNARK that does well on `secp256k1` is a question that the industry has been aware of, but it is a difficult one. In fact, a related question, verifying ECDSA signatures over `ed25519` for zk-Bridge with Solana, is harder, and it is often regarded as *the pearl of SNARK* today, as, even with hardware acceleration, there are very few algorithms that do well [BCHO22],

1.5 Rest of the paper

The rest of the paper is organized as follows:

- Section 2: preliminaries on the cryptographic building blocks
- Section 3: design rationale, optimization used in production, and an informal security proof for the compiler from the inspection model to NIZK
- Section 4: batching and an informal security proof for delegated Schnorr
- Section 5: implementations and benchmarks
- Section 6: some open problems of interest

And in the appendix there are:

- Appendix A: indexer, prover, and verifier of a binary-testing-friendly variant of TurboPlonk used in the implementation and benchmark, for completeness

- Appendix B: (im)possibility results about Zerocash with only access to a deterministic ECDSA signing oracle, which shows the limitations of the techniques presented in this paper

1.6 Bug bounty program

This paper and its implementation are covered by the $1M bug bounty program through a grant from Findora Foundation. To report a bug, email Findora Cryptography Team crypto@findora.org. The amount of the award will be determined by the Findora Foundation based on its severity. It should generally fall in the range of $500 to $10,000. The bug bounty program is not restricted to security bugs and covers suggestions for improvement as well.

In addition, readers who are interested in working on some research problems related to this paper (see Section 6 for some examples) can consider applying for a Findora Foundation research grant. To start a discussion, email Findora Cryptography Team crypto@findora.org.

2 Cryptographic building blocks

In this section we provide some background on the cryptographic tools that we use in this paper.

Fiat-Shamir transform. The Fiat-Shamir transform [FS86], also known as Fiat-Shamir heuristic, provides a way to convert a public-coin interactive proof of knowledge with an honest verifier to a non-interactive zero-knowledge proof of knowledge (NIZKoK). The high-level idea is to replace the verifier's challenges with the output from the random oracle over the ongoing transcript, and we expect that such challenges are hard enough such that the prover does not have an advantage.

It is important to note that the Fiat-Shamir transform is a heuristic because there are counterexamples [GT03; Bit+13]. This is similar to the counterexamples for random oracles [CGH98; BG81]. Despite these negative results, for most protocols used in practice, it is generally assumed that we do not know of a concrete attack that leverages the Fiat-Shamir transform to break the non-interactive protocol. Indeed, many cryptographic protocols used in production today rely on the Fiat-Shamir transform, such as Chaum-Pedersen [CP92], Schnorr [Sch89], Bulletproofs [BBBPWM18], and TurboPlonk [GWC19] that we will describe below.

Schnorr protocol. The Schnorr protocol [Sch89] is a classic example of the use of the Fiat-Shamir transform, which transforms the Schnorr identification protocol to the Schnorr signature. Such a signature, first of all, proves the knowledge of the corresponding secret key, and secondly, due to the use of Fiat-Shamir transform, it is also tied to a specific message, which provides unforgeability in the context of a digital signature scheme.

Chaum-Pedersen protocol. The seminal work by Chaum and Pedersen [CP92] presents a protocol that proves discrete log equality for two groups of the same order, which is a direct application of the Fiat-Shamir transform and can be considered as an extension to the Schnorr protocol. The motivation of this paper, as described in Section 1, is to generalize the Chaum-Pedersen protocol to the setting where the two groups may have different orders.

Bulletproofs. Bulletproofs [BBBPWM18] is a very special proof system with transparent setup. Unlike many other zk-SNARKs with transparent setup, Bulletproofs has a small proof size like its universal-setup peers. And, unlike many of its universal-setup peer, Bulletproofs does not use pairing, but instead is based on inner-product arguments, which can work with non-pairing-friendly curves such as `curve25519` and `secq256k1`. The main limitation of Bulletproofs, also the limitation of transparent-setup inner-product arguments, is that the proof generation time is comparably higher, and the proof verification time goes linearly with the size of the statement being proven. Nevertheless, Bulletproofs is particularly useful in proving small statements.

TurboPlonk. The state-of-the-art proof systems for many blockchain applications belong to the Plonk family [GWC19], which include Plonk, TurboPlonk (which uses customized gates), and UltraPlonk (which additionally uses lookup). Generally, people can use customized gates to increase the sparsity of some polynomials involved in the computation, as well as reduce the length of some dense polynomials involved in the computation, and therefore reduce the proving time by having fewer multiscalar multiplication operations. This is sometimes referred to as "the sparsity theory" of polynomial interactive oracle proof (PIOP). Different variants of Plonk often leverage two or three of the common tools for PIOP: (1) sumcheck, (2) the shifting trick, and (3) universal hashing, and can yield a better performance than many R1CS-based proof systems. TurboPlonk has a universal setup, and its proof size is short so that it is suitable for blockchain applications. Note that Plonk can also refer to its arithmetization. For example, Halo2 [Hal] uses Plonk-style arithmetization, but its polynomial commitment scheme uses inner-product arguments, which is closer to Bulletproofs.

Nonnative field. Many zero-knowledge proof systems are tailored to doing proofs over statements described in a specific field. For example, pairing-based SNARKs often are defined over the scalar field of that pairing-friendly curve. This becomes an issue when interoperability in different fields is needed in the statement. For example, in the two applications described in Section 1, the Zerocash construction is largely defined over the BLS12-381 curve due to various reasons described before, but it needs to reason about other field elements over the scalar field of `curve25519` and/or `secq256k1`. To handle this, the standard solution today is to use nonnative fields [KPS18; OWWB20] to simulate the computation on a prime field over a different prime field. The high-level idea is to split an element on one prime field into several limbs on another prime field and perform addition and multiplication over such limbs while making sure that they do not overflow, which is done by expensive simulation of modular reduction.

3 The inspection model for zero-knowledge proofs

In this section we present design rationale, optimization used in production, and an informal security proof for this inspection model and its compiler. We also provide an alternative construction.

3.1 Design rationale

We now describe the design rationale behind the inspection model.

Inspector. The motivation of the inspection model comes from the saddening reality that the Schnorr protocol, as a proof of knowledge is, however, just a proof of knowledge. Though it can show that the prover knows the discrete log, the protocol itself does not prove anything else.

This challenge will be trivial, however, if we drop the zero-knowledge property, since the prover can simply disclose the discrete log. This is also the case for many other protocols, and the battle between (knowledge) soundness and zero-knowledge has always been here. It would be ideal, when designing the protocol, that there is a "demilitarized zone" (DMZ) where such a battle does not happen. This is where the "inspector" $\mathbf{i}$ shows up.

Defining the inspector. When defining the inspector $\mathbf{i}$, we want to make the definitions as simple and minimal as possible. That is, inspector $\mathbf{i}$'s interactions with prover $\mathbf{P}$ and verifier $\mathbf{V}$ should be sufficient for different kinds of inspection, nothing else. This minimalism principle is reflected in our design in the following ways:

- Before the decision round, inspector $\mathbf{i}$ remains silent. Though it receives secret messages from the prover $\mathbf{P}$, inspector $\mathbf{i}$ does not need to respond to them.
- Before the decision round, inspector $\mathbf{i}$ hears additional information from prover $\mathbf{P}$, but not from verifier $\mathbf{V}$. This restriction is added because we focus on protocols that can easily be made non-interactive through Fiat-Shamir transform, which are public-coin, in which case all messages from verifier $\mathbf{V}$ can be deduced from the transcript. There is no other information that verifier $\mathbf{V}$ needs to share with inspector $\mathbf{i}$.
- At the decision round, inspector $\mathbf{i}$ can only send out a veto signal $\texttt{VETO}$ or $\perp$, and if prover $\mathbf{P}$ is honest, $\mathbf{i}$ for sure will not veto. We restrict the inspector's message to verifier $\mathbf{V}$ this way for the following reasons. Note that inspector $\mathbf{i}$ has to pass some information in the protocol to verifier $\mathbf{V}$, otherwise it has no effect on the outcome. However, this message may bring challenges to the zero-knowledge property, as inspector $\mathbf{i}$ could leak information to verifier $\mathbf{V}$ about those secret messages that prover $\mathbf{P}$ only shares with the inspector $\mathbf{i}$. To solve this problem, the easiest way is to restrict the information that inspector $\mathbf{i}$ can share with verifier $\mathbf{V}$, for example, one bit. Note that even one bit can be leaky, but in our definitions, when prover $\mathbf{P}$ is honest, this bit will always be false (i.e., "$\perp$" in our case), and therefore can be easily simulated by a simulator, which immediately preserves the zero-knowledge property.

To summarize, inspector $\mathbf{i}$ hears additional information from $\mathbf{P}$ but not $\mathbf{V}$, and the only information that $\mathbf{i}$ can say is one bit in the decision round to $\mathbf{V}$, and this bit is trivial ("$\perp$") for a valid proof.

Instantiating the inspector. It is not difficult to add a new entity to a cryptographic protocol, but for it to be useful, it must be instantiated with concrete cryptographic primitives in the real world. In particular, there is *no inspector* in the real world.

However, think about the inspector as a special case of an oracle in the interactive proof system. It is not uncommon that we build an interactive proof protocol that uses some oracles, and replace them during the Fiat-Shamir transform.

A concrete example is polynomial interactive oracle proofs (PIOPs) that are behind a number of modern proof systems including Sonic [MBKM19], Marlin [CHMMVW20], Gemini [BCHO22], and Plonk [GWC19]. In that framework, in the interactive protocol, the polynomials are treated as oracles, and the verifier has holographic and virtual access to these oracles that are created by the

prover, and the verifier can query the polynomial on some points. Such polynomial oracles clearly do not exist in the real world, but they can be replaced, during the Fiat-Shamir transform, with a concrete instantiation of a polynomial commitment scheme.

For the inspection model, we replace it with a protocol that is a combination of simulatable commitments [MP03] and NIZK, described as follows.

First, we note that in the decision round, the inspector $\mathbf{i}$ can only send out one bit, and if the proof is valid, this bit is trivial. This is in essence similar to NIZK verification, as verifier $\mathbf{V}$ outputs $b \in \{0, 1\}$, and for a valid proof, the bit is always $b = 1$, which is trivial. Therefore, we can run the inspector $\mathbf{i}$'s decider in NIZK, which proves that the decision is to not veto.

Second, it is not enough for us to put the decider into NIZK, because we also need to force the prover $\mathbf{P}$ to send the secret message for a specific round, for example, $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ for the i-th message, before it receives the message sent by verifier $\mathbf{V}$, which, in the public-coin setting, is the challenge as in $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$. This requires some sort of simulatable commitment [MP03] with extractability, and it can be done with hash commitment. Particularly, whenever it is the time for a prover $\mathbf{P}$ to send $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ to $\mathbf{i}$, we replace it with a commitment c_i (with randomizer r_i) that will be sent to the verifier $\mathbf{V}$ instead. This is in essence similar to the setting where the inspector $\mathbf{i}$'s decider asks $\mathbf{V}$ to receive those messages on its behalf and the prover $\mathbf{P}$ is "encrypting" those messages.

The commitment c_i can be described as follows:

$$c_i = H(\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}; r_i)$$

This commitment is later opened by the inspector $\mathbf{i}$'s decider, given r_i provided by prover $\mathbf{P}$ when generating the NIZK. To sum up, by a commitment scheme we enforce the timing of those secret messages, and by NIZK we can run a decider in zero knowledge.

3.2 Optimization used in production

We now present some optimization techniques that we use in our implementation.

SNARK-friendly simulatable commitment with extractability. In production, the NIZK can be instantiated with zk-SNARK, and this kind of proof system often prefers doing computation over a specific field $\mathbb{F}$. Therefore, an optimization is to use a SNARK-friendly simulatable commitment schemes that have extractability, such as Rescue [AABSDS20] over this specific field $\mathbb{F}$. It is important for us to emphasize the field because, for example, in our application on Zerocash with `secp256k1` keys, the field $\mathbb{F}$ that the hash function will be defined over is BLS12-381's scalar field, while the field elements that we commit are on the scalar field of `secq256k1`. Some sort of adaptation is needed to reconcile such differences, such as storing the nonnative element as limbs, but what is important here is to use Rescue for $\mathbb{F}$ in zk-SNARK.

Omitting unnecessary public messages. As our examples in Section 1 show, although the compiler provides all $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ and $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ as the instance to the NIZK that runs the inspector $\mathbf{i}$'s decider, some of the public messages, $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$, or some parts of them, may not be of the decider's interest, and therefore such instances can be omitted from the NIZK proof generation and verification.

Preprocessing of the NIZK instance. For NIZK there can be computation over the public instance $\mathbb{x}$ that should better be performed outside. In our examples in Section 1.4, instead of passing:

$$(c_1, \beta, \lambda, s_1, s_3, s_5)$$

The NIZK uses the following instance:

$$\big(c_1, \beta, \lambda, \beta\lambda, \lambda^2, \beta\lambda^2, s_1 + \lambda s_3 + \lambda^2 s_5\big)$$

in which the instance includes $(\beta\lambda, \lambda^2, \beta\lambda^2)$, which can actually be derived already from β and λ, and it includes $s_1 + \lambda s_3 + \lambda^2 s_5$, which can be derived from λ, s_1, s_3, s_5. The reason for such preprocessing is that all these elements above are indeed over the scalar field of `secq256k1` and not over the scalar field of BLS12-381, so deriving these numbers through nonnative fields is expensive. To reduce the overhead, some precomputation is pushed to the instance, and it expects verifier $\mathbf{V}$ to do the precomputation. Note that the omission of unnecessary public messages, which we describe above, is a special case of preprocessing. There is some overlapping.

3.3 Informal security proof

We now present an informal security proof, in which we strive to avoid the discussion about Fiat-Shamir transform. Our proof strategy is to first show that the interactive protocol between $\mathbf{P}, \mathbf{i}, \mathbf{V}$ is equivalent to another interactive protocol between $\mathbf{P}$ and $\mathbf{V}$. Then, one can straightforwardly apply the Fiat-Shamir transform. The final result of this Fiat-Shamir transform is the output of our compiler for the original interactive protocol between $\mathbf{P}, \mathbf{i}, \mathbf{V}$.

This informal proof will focus on zero-knowledge and knowledge soundness.

The inspector vanishes. We now process each round and gradually remove the inspector $\mathbf{i}$ from the interactive protocol.

• **Proof generation:** For the i-th round:

1. "$\mathbf{P}$ *sends a message* $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ *to* $\mathbf{i}$" will be processed as follows:
   (a) Sample a random blinding factor r_i
   (b) Compute a commitment $c_i := H(\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}; r_i)$
   (c) Send c_i to verifier $\mathbf{V}$
2. "$\mathbf{P}$ *sends a message* $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ *to* $\mathbf{i}$ *and* $\mathbf{V}$" will be processed as follows:
   - Send $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ to verifier $\mathbf{V}$
3. "$\mathbf{V}$ *sends a message* $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ *to* $\mathbf{P}$ *and* $\mathbf{i}$" will be processed as follows:
   - Send $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ to prover $\mathbf{P}$

Then, for the decision round, prover $\mathbf{P}$ directly generates an NIZK:

1. Generate an NIZK π with the following instance $\mathbb{x}$, witness $\mathbb{w}$, and statement Π
   - Instance $\mathbb{x}$: all the $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$, all the $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$, and all the c_i
   - Witness $\mathbb{w}$: all the $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$, and all the r_i

- Statement Π:
  - Check for all i: $c_i \stackrel{?}{=} H(\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}; r_i)$
  - Check that $\mathbf{i}([\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}, \mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}, \mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}]_i) \stackrel{?}{=} \perp$

2. Send π to verifier $\mathbf{V}$

• **Proof verification:** For the i-th round,

1. *"$\mathbf{P}$ sends a message $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ to $\mathbf{i}$"* will be processed as follows:
   - Put c_i into the transcript
2. *"$\mathbf{P}$ sends a message $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ to $\mathbf{i}$ and $\mathbf{V}$"* will be processed as follows:
   - Put $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$ into the transcript
3. *"$\mathbf{V}$ sends a message $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ to $\mathbf{P}$ and $\mathbf{i}$"* will be processed as follows:
   - Do nothing, since $\mathbf{V}$ is public-coin, $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$ can be computed from the transcript, through the Fiat-Shamir transform

Then, for the decision round, verifier $\mathbf{V}$ additionally checks the NIZK:

- Verify the NIZK π using the instance—all the $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}$, all the $\mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}$, and all the c_i—which come from the proof or from the transcript via Fiat-Shamir transform
- Check that $\mathbf{V}([\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i},\mathbf{V}}, \mathsf{msg}^i_{\mathbf{V}\to\mathbf{P},\mathbf{i}}]_i) \stackrel{?}{=} \texttt{ACCEPT}$

Honest-verifier zero-knowledge. Since the original protocol is honest-verifier zero-knowledge (HVZK), there is a simulator $\mathbf{S}$ that can simulate the honest verifier's view. We now show that we can easily construct another simulator $\mathbf{S}'$ for the new interactive protocol above.

First of all, since we use simulatable commitments [MP03], $\mathbf{S}'$ can simulate all the commitments c_i that commit the secret messages to the inspector $\mathbf{i}$.

Then, $\mathbf{S}'$ invokes the simulator for the NIZK. Note that our NIZK might need to work with some oracles, such as a random oracle, if the simulatable commitment uses it (as in the case of hash commitment). We assume that despite such nuances, we are able to use this simulator.

Knowledge soundness. Recall that the compiler uses simulatable commitment with extractability, which is, fortunately, not difficult to instantiate in the real world. The use of a simulatable commitment scheme with extractability means that the extractor can still effectively extract $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ even if they are no longer being sent out, by using the extractor of this commitment scheme.

Note that we do not need NIZK to have extractability, since the witness that we plan to extract is already in the commitments that we can extract.

3.4 Alternative construction

The construction we discuss in this paper uses a simulatable commitment with extractability and an NIZK. As a byproduct of the knowledge soundness proof above, we note that one can also use a simulatable commitment and an extractable NIZK, because $\mathsf{msg}^i_{\mathbf{P}\to\mathbf{i}}$ can also be extracted through the extractable NIZK.

We favor a simulatable commitment with extractability because the hash commitment, in the random oracle model, easily achieves the requirements, and can be instantiated quite efficiently in practice using SNARK-friendly primitives.

This requirement, however, can be challenging for some NIZK constructions. For example, for Marlin [CHMMVW20] to have extractability, one needs to either work with the algebraic group model or assume the Knowledge of Exponent Assumption (KEA) and endure an overhead of $2\times$ for polynomial commitments.

4 Delegated Schnorr protocol

In this section we describe how to do batching in the delegated Schnorr protocol and provide an informal security proof.

4.1 Batching

We now present a batched version of delegated Schnorr protocol with the following performance numbers for doing a batch of K commitments:

- NIZK performs only one (instead of K) post-multiplication modular reduction in nonnative fields, or informally, one nonnative mul. The amount of the computation in NIZK should grow slowly with K, but its impact on proof size, proof generation time, and proof verification time depends on the concrete proof system.
- Outside NIZK, there are $(2 + K)$ (instead of $3K$) elements in $\mathbb{F}$ and K elements in $\mathbb{G}$. In terms of computation, it performs $(2K + 1)$ (instead of $3K$) scalar-point muls.

Opportunities for batching. There are several opportunities for batching. We first discuss the main one. Note that in NIZK, it is supposed to check the following equations, each of which takes one post-multiplication modular reduction.

$$\begin{aligned} s_1 &= \beta x + a \\ s_3 &= \beta y + c \\ s_5 &= \beta z + e \end{aligned}$$

However, after verifier $\mathbf{V}$ receives s_1, s_3, s_5 from prover $\mathbf{P}$, it can check a random linear combination of equations above. In particular, verifier $\mathbf{V}$ can choose a random challenge $\lambda \leftarrow\$ \mathbb{F}$, and check the following instead:

$$\begin{pmatrix} s_1 \\ +\lambda s_3 \\ +\lambda^2 s_5 \end{pmatrix} = \beta \begin{pmatrix} x \\ +\lambda y \\ +\lambda^2 z \end{pmatrix} + \begin{pmatrix} a \\ +\lambda c \\ +\lambda^2 e \end{pmatrix}$$

But, it does not seem that the amount of nonnative field operations goes down as multiplying by λ itself incurs similar overhead. This is avoided through preprocessing. Note that the equation above

can be rewritten as:

$$\begin{pmatrix} s_1 \\ +\lambda s_3 \\ +\lambda^2 s_5 \end{pmatrix} = \begin{pmatrix} \beta x \\ +\beta\lambda y \\ +\beta\lambda^2 z \end{pmatrix} + \begin{pmatrix} a \\ +\lambda c \\ +\lambda^2 e \end{pmatrix}$$

and we now highlight all elements that can be computed during the preprocessing:

$$\begin{pmatrix} s_1 \\ +\lambda s_3 \\ +\lambda^2 s_5 \end{pmatrix} = \begin{pmatrix} \beta\, x \\ +\,\beta\lambda\, y \\ +\,\beta\lambda^2\, z \end{pmatrix} + \begin{pmatrix} a \\ +\,\lambda\, c \\ +\,\lambda^2\, e \end{pmatrix}$$

As one can see, all the secret values x, y, z, a, c, e will only be multiplied by one other value. In addition, in nonnative fields, the intermediate representations of $\beta x, \beta\lambda y, \beta\lambda^2 z, \lambda c, \lambda^2 e$ do not need to each go through an individual post-multiplication modular reduction. In fact, the NIZK can add these intermediate representations together and then perform one modular reduction. This is why, regardless of K, the number of post-multiplication modular reductions is only one.

Now we turn our attention to the non-NIZK part.

There are still K elements in $\mathbb{G}$, which are the blinding elements used in the Schnorr protocol. What we can save is the number of elements in $\mathbb{F}$, from $3K$ to $2 + K$. To see why this is possible, let us think about what the three field elements for each equation are:

- The hash commitment for a message to the inspector, e.g., $c_1 = H(x, a; r_1)$
- A response scalar s_1
- Another response scalar s_2

Our first observation is that one hash commitment is capable of committing all the messages from each of the K equations. That is,

$$c_1 = H(x, a, y, c, z, e; r_1)$$

The second observation is that, for the response scalars at the even indices, such as s_2, s_4, s_6, we do not actually care, as we are not proving knowledge about the randomizers. Therefore, some readers may already discover this from our two applications in Section 1, that our response scalars in the interactive protocol are sent in two different rounds.

- In the second round, only response scalars at the odd indices, such as s_1, s_3, s_5, are sent.
- After receiving a challenge λ from verifier $\mathbf{V}$, the response scalars at the even indices will be sent, but as a random linear combination:

$$s_2 + \lambda s_4 + \lambda^2 s_6$$

Therefore, we can indeed merge all the response scalars at the even indices together. This means that in a batched protocol, the $(2+K)$ field elements include one hash commitment, one combined response scalar for those at the even indices, and K response scalars at the odd indices.

Finally we want to show why the number of scalar-point multiplication operations outside NIZK is $(2K+1)$ instead of $3K$. Note that previously, there are K equations to check, and each equation requires three scalar-point multiplication operations, in total $3K$.

$$\begin{aligned} s_1G + s_2H &= \beta P + C_1 \\ s_3G + s_4H &= \beta Q + C_2 \\ s_5G + s_6H &= \beta R + C_3 \end{aligned}$$

Note that this can be batched using a random linear combination of λ, and in fact this is mandatory now because the proof only has $s_2 + \lambda s_4 + \lambda^2 s_6$. We now have the following equation to check:

$$\begin{pmatrix} s_1 \\ +\lambda s_3 \\ +\lambda^2 s_5 \end{pmatrix} G + \begin{pmatrix} s_2 \\ +\lambda s_4 \\ +\lambda^2 s_6 \end{pmatrix} H = \begin{pmatrix} \beta P \\ +\beta\lambda Q \\ +\beta\lambda^2 R \end{pmatrix} + \begin{pmatrix} C_1 \\ +\lambda C_2 \\ +\lambda^2 C_3 \end{pmatrix}$$

And we now highlight the scalars.

$$\begin{pmatrix} s_1 \\ +\lambda s_3 \\ +\lambda^2 s_5 \end{pmatrix} G + \begin{pmatrix} s_2 \\ +\lambda s_4 \\ +\lambda^2 s_6 \end{pmatrix} H = \begin{pmatrix} \beta P \\ +\beta\lambda Q \\ +\beta\lambda^2 R \end{pmatrix} + \begin{pmatrix} C_1 \\ +\lambda C_2 \\ +\lambda^2 C_3 \end{pmatrix}$$

The total number of scalar-point multiplication operations is therefore:

$$1 + 1 + K + (K-1) = 2K + 1$$

Put it together. We now present the general algorithms to perform the delegated Schnorr protocols over K Pedersen commitments. We consider each commitment P_i as follows:

$$P_i = x_i G + \gamma_i H$$

We now describe the batched protocol.

• **Proof generation:**

1. Sample $a_i, b_i \leftarrow\$ \mathbb{F}_G$, compute $C_i := a_iG + b_iH$ for $i \in \{1, 2, ..., K\}$; sample $r_1 \leftarrow\$ \mathbb{F}_N$, compute $c_1 := H([(x_i, a_i)]_1^K; r_1)$, and put $(c_1, [C_i]_1^K)$ into the transcript
2. Compute the challenge $\beta \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
3. Compute $s_{2i-1} := \beta x_i + a_i$ and put them into the transcript
4. Compute the challenge $\lambda \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
5. Compute $s_{\mathsf{even}} := \sum_{i=1}^K \lambda^{i-1}(\beta\gamma_i + b_i)$
6. Set block B to be $(c_1, [C_i]_1^K, [s_{2i-1}]_1^K, s_{\mathsf{even}})$
7. Append the following statement (block A) into NIZK:
   - Additional instance $\mathbb{x}^+$: c_1, $[C_i]_1^K$, β, $[s_{2i-1}]_1^K$, s_{even}, and the instance for precomputation, $[\beta\lambda^{i-1}]_1^K$, $[\lambda^{i-2}]_2^K$, and $\sum_{i=1}^K \lambda^{i-1} s_{2i-1}$

- Additional witness $\mathbb{w}^+$: $[x_i]_1^K$, $[a_i]_1^K$, r_1
- Additional statement Π^+:
  - Check $c_1 = H([(x_i, a_i)]_1^K; r_1)$
  - Check $(\sum_{i=1}^K \lambda^{i-1} s_{2i-1}) = \sum_{i=1}^K (\beta\lambda^{i-1}) \cdot x_i + a_1 + \sum_{i=2}^K (\lambda^{i-2}) \cdot a_i$

8. Proceed with the rest of the protocol

• **Proof verification:**

1. Parse block B as $(c_1, [C_i]_1^K, [s_{2i-1}]_1^K, s_{\mathsf{even}})$
2. Compute the challenge $\beta, \lambda \in \mathbb{F}_G$ from the transcript through the Fiat-Shamir transform
3. Check that $\sum_{i=1}^K (\beta\lambda^{i-1} \cdot P_i + \lambda^{i-1} \cdot C_i) = (\sum_{i=1} \beta\lambda^{i-1} s_{2i-1}) \cdot G + s_{\mathsf{even}} \cdot H$
4. Check the NIZK proof with the additional instance:

$$\mathbb{x}^+ := (c_1, \beta, [\beta\lambda^{i-1}]_1^K, [\lambda^{i-2}]_2^K, \textstyle\sum_{i=1}^K \lambda^{i-1} s_{2i-1})$$

4.2 Informal security proof

Though one can already prove the security of this protocol by applying the compiler, here we provide a standalone informal security proof on the resultant protocol. For simplicity, we present a proof in which the delegated Schnorr protocol only opens one group element.

Soundness. Assume that prover $\mathbf{P}$ has a strategy that can mislead verifier $\mathbf{V}$ to accept $P = xG + \gamma H$, but $\Pi(x) = \texttt{false}$. Given the soundness of NIZK and the binding property of commitment, we can assume $x' \neq x$ is the value committed in c_1 and it holds that $\Pi(x') = \texttt{true}$. We know that:

$$c_1 = H(x', a'; r_1)$$

for some a'. Let us say that $C_1 = aG + bH$. The equation $s_1 G + s_2 H = \beta P + C_1$ implies that:

$$\beta x + a = \beta x' + a'$$

which implies that the malicious prover $\widetilde{\mathbf{P}}$ must set:

$$a' = \beta(x - x') + a$$

However, β is known only after a' is committed. Except with negligible probability, the malicious prover $\widetilde{\mathbf{P}}$ can only do so with $x = x'$, which is a contradiction. This result also makes it easy to show knowledge soundness, as one can either extract from the simulatable commitment if it has extractability, or from NIZK if it has extractability.

Zero-knowledge. For an honest prover who follows the protocol, invoking the simulator for NIZK, the view of the verifier after seeing the proof is:

$$c_1, C_1, s_1, s_2$$

Note that the commitment has the hiding property. We can now focus on C_1, s_1, s_2, and we want to show that there is a simulator that can simulate such values that satisfy $s_1 G + s_2 H = \beta P + C_1$

with β depends on c_1 and C_1. This proof now requires a restricted programmable random oracle to manipulate the Fiat-Shamir transform. The simulator samples β, s_1, s_2 and sets:

$$\widetilde{C_1} := -\beta P + s_1 G + s_2 H$$

and programs the random oracle to output β when c_1 and $\widetilde{C_1}$ appear on the transcript.

5 Implementation and evaluation

In this section we elaborate on the implementation in Findora's Noah library and present some benchmark results on the performance of the construction presented in this paper and the breakdown.

5.1 Implementation

We now describe our implementation of the two applications (Maxwell-Zerocash switching and Zerocash with secp256k1 keys) layer-by-layer. Most code in this implementation comes from either the existing Noah library [Noa] or third-party libraries such as Dalek's `bulletproofs` [Dala], `x25519-dalek` [Dalc], and `curve25519-dalek` [Dalb]. This allows us to base our implementation on code that has been used for a while and has undergone either informal or formal auditing.

Cryptographic building blocks. The Noah library [Noa] uses the `arkworks-rs` library [ark] as the backend since its code has been used by several ZK companies and is production-ready. One of the main building blocks in our applications is Bulletproofs. For Maxwell-Zerocash switching, we use Dalek's `bulletproofs` off the shelf. For Zerocash with `secp256k1` keys, we port Dalek's `bulletproofs` to use `arkworks-rs` as a backend, and then we fill in the field/curve parameters for the `secp256k1` and `secq256k1` curves.[5]

Delegated Schnorr. Recall that the delegated Schnorr protocol consists of two parts: block A, which is inside NIZK, and block B, which is outside. Our Rust implementation of this protocol is in the Noah library. For block B, we implement it in a group-agnostic manner, which would be useful for our two applications: Maxwell-Zerocash switching is over the Ristretto group over `curve25519`, while Zerocash with `secp256k1` keys is over `secq256k1`. To prove the statement in block A efficiently, we use a binary-testing-friendly TurboPlonk described in Appendix A, which helps reduce the overhead of nonnative fields. To do this, we use standard techniques in [KPS18] and [OWWB20].

Application: Maxwell-Zerocash switching. We implement a protocol for Maxwell-to-Zerocash switching and a protocol for Zerocash-to-Maxwell switching in the Noah library. Both are capable of hiding the amount and the asset type, following the design presented in Section 1.3.

[5] As we mentioned previously in Section 1, we also implemented `bulletproofs` for another curve called Canaan that also embeds `secp256k1`, which would be useful if hashing to curve is needed. Its implementation can be found in https://github.com/FindoraNetwork/ark-bulletproofs.

Application: Zerocash with secp256k1 keys. The Findora blockchain network has moved to use `secp256k1` in the smart contracts on EVM chain, the Maxwell transactions on the UTXO chain, and the Zerocash transactions on the UTXO chain. For the Zerocash transactions, we focus on the case in which all inputs belong to the same public key.[6] Therefore, for each transaction, we only need one Bulletproofs to prove a scalar multiplication. TurboPlonk reads the input commitments of the Bulletproofs to obtain a public key and a secret key, with the assurance that Bulletproofs has already proven the scalar multiplication relation, and uses these keys for the rest of the Zerocash protocol, particularly for the computation of the nullifiers.

Source code. Readers can find the source code of the implementation in:

https://github.com/FindoraNetwork/noah

5.2 Evaluation setup

For the two applications, Maxwell-Zerocash switching and Zerocash with `secp256k1` keys, users' machines, instead of some servers, will be running the prover. This is due to the needs of privacy—a user cannot easily delegate the proof generation to an untrusted third party. Therefore, we choose to run the benchmark on the following two representative environments.

- `m5.xlarge`, Intel Xeon Platinum 8175M CPU at 2.50 GHz with 4 vCPU and with 16 GB memory
- `mac2.metal`, Apple's M1 chip with 8 CPU cores with 3.2 GHz, with 16 GB memory

5.3 Evaluation on Maxwell-Zerocash switching

We benchmark Maxwell-Zerocash switching on these two setups. We focus on (1) the number of gates in TurboPlonk, (2) proof generation time, (3) proof size, and (4) proof verification time.

Number of gates in TurboPlonk. We present a breakdown of the number of gates in TurboPlonk proofs in Table 6 and Table 7. It is expected that the composition and the total will be very different between Maxwell-to-Zerocash and Zerocash-to-Maxwell, for the following reasons:

- In Maxwell-to-Zerocash, the prover only needs to assemble a new commitment, while in Zerocash-to-Maxwell, the prover needs to prove the existing commitment is in the global Merkle tree.
- In Maxwell-to-Zerocash, the prover does not need to prove the knowledge of the secret key using the delegated Schnorr protocol, because in the Maxwell construction, the address as well as the public key is public. However, in Zerocash-to-Maxwell, the public key of the Zerocash transaction also needs to be hidden, and therefore the use of delegated Schnorr protocol over `secq256k1` is not avoidable.

[6] This restriction can be lifted without a significant overhead for block A of the delegated Schnorr protocol using batching. To do so, prover $\mathbf{P}$ does a Bulletproofs for each public key in the inputs, and the delegated Schnorr protocol opens a linear combination of the input commitments of these Bulletproofs.

Proof generation time. We measure the overall proof generation time and present a breakdown that consists of the time to generate the TurboPlonk proof, or the time to generate each delegated Schnorr proof. Below we show the single-thread and multithread performance.[7]

- For Maxwell-to-Zerocash:
  - Proof generation in total takes 1.015 s on `m5.xlarge` and 0.638 s on `mac2.metal`, and in the case of multithreading, 0.606 s on `m5.xlarge` and 0.285 s on `mac2.metal`
  - The TurboPlonk proof takes 1.012 s on `m5.xlarge` and 0.635 s on `mac2.metal`, and in the case of multithreading, 0.603 s on `m5.xlarge` and 0.283 s on `mac2.metal`
  - The Ristretto delegated Schnorr proof takes. 0.003 s on `m5.xlarge` and 0.002 s on `mac2.metal`
- For Zerocash-to-Maxwell:
  - Proof generation in total takes 5.53 s on `m5.xlarge` and 3.58 s on `mac2.metal`, and in the case of multithreading, 3.78 s on `m5.xlarge` and 2.01 s on `mac2.metal`
  - The TurboPlonk proof takes 3.08 s on `m5.xlarge` and 1.93 s on `mac2.metal`, and in the case of multithreading, 1.95 s on `m5.xlarge` and 0.95 s on `mac2.metal`
  - The delegated Schnorr proof over Ristretto takes 0.003 s on `m5.xlarge` and 0.002 s on `mac2.metal`
  - The delegated Schnorr proof over `secp256k1` takes 0.005 s on `m5.xlarge` and 0.003 s on `mac2.metal`
  - Bulletproofs over `secq256k1` takes 2.31 s on `m5.xlarge` and 1.55 s on `mac2.metal`, and in the case of multithreading, 1.69 s on `m5.xlarge` and 0.97 s on `mac2.metal`

We can see that the proof generation of TurboPlonk and Bulletproofs takes most of the time.

Proof size. We now present the proof sizes and their breakdown. For the Maxwell-to-Zerocash switching, the proof has 1456 bytes (1.4 KB), which consists of:

- a delegated Schnorr proof over Ristretto of 296 bytes (20.3%)
- a TurboPlonk proof of 1160 bytes (79.7%)

For the Zerocash-to-Maxwell switching, the proof has 3279 bytes (3.2 KB), which consists of:

- a delegated Schnorr proof over Ristretto of 296 bytes (9.0%)
- a delegated Schnorr proof over `secq256k1` of 419 bytes (12.8%)
- three Bulletproofs input commitments of 131 bytes (4.0%)
- a Bulletproofs over `secq256k1` of 1273 bytes (38.8%)
- a TurboPlonk proof of 1160 bytes (35.4%)

Proof verification time. Similarly, we measure the proof verification time as well as the breakdown for each different type of proof. Here we only present single-thread numbers since the verifier here is not made parallel.

- For Maxwell-to-Zerocash:
  - Proof verification in total takes 0.013 s on `m5.xlarge` and 0.009 s on `mac2.metal`

[7] The multithread performance may be improved in the future since, at this moment, only the multiscalar multiplication (MSM) is made parallel.

	# gates
Witness validity	322 (24.0%)
Commitment opening	306 (22.9%)
Ristretto delegated Schnorr	711 (53.1%)
Total	1339

Table 6: Maxwell to Zerocash.

	# gates
Witness validity	3888 (48.8%)
Commitment opening	306 (3.8%)
Ristretto delegated Schnorr	711 (8.9%)
`secq256k1` delegated Schnorr	3058 (38.4%)
Total	7963

Table 7: Zerocash to Maxwell.

  - Verification of the TurboPlonk proof takes 0.013 s on `m5.xlarge` and 0.009 s on `mac2.metal`
  - Verification of the delegated Schnorr proof over Ristretto takes 0.0004 s on `m5.xlarge` and 0.0004 s on `mac2.metal`
- For Zerocash-to-Maxwell:
  - Proof verification in total takes 0.24 s on `m5.xlarge` and 0.16 s on `mac2.metal`
  - Verification of the TurboPlonk proof takes 0.013 s on `m5.xlarge` and 0.009 s on `mac2.metal`
  - Verification of the delegated Schnorr proof over Ristretto takes 0.0004 s on `m5.xlarge` and 0.0003 s on `mac2.metal`
  - Verification of the delegated Schnorr proof over `secq256k1` takes 0.001 s on `m5.xlarge` and 0.001 s on `mac2.metal`
  - Verification of the Bulletproofs over `secq256k1` takes 0.224 s on `m5.xlarge` and 0.149 s on `mac2.metal`

We note that the dominating cost is the verification of the Bulletproofs. This is not surprising, as Bulletproofs has a linear verification time. However, this is not without solutions, as Bulletproofs can be very efficiently batched together. Batching, however, needs to be done carefully. As if one of the proofs is incorrect, we do not know which one is. However, with some care, we can still leverage the batching. In a blockchain network where a group of validators decide how the ledger progresses. One validator, as the proposer, needs to verify such proofs one by one, while the rest of the validators, who vote, can simply check the proofs that the proposer approves in a batch.

5.4 Evaluation on Zerocash with secp256k1 keys

Similarly, we focus on (1) the number of gates in TurboPlonk, (2) proof generation time, (3) proof size, and (4) proof verification time.

Number of gates in TurboPlonk. We present the number of gates in TurboPlonk and its breakdown in Table 8 and Table 9. Since we require all the inputs to come from the same address, the amount of work in `secq256k1` does not depend on the number of inputs or outputs. We can also see that as the number of inputs and outputs increases, the overhead of witness validity and asset equality increases. The witness validity, which includes the Merkle tree proof, is expected to increase linearly, while the asset equality is expected to grow quadratically.

Proof generation time. We again measure the overall proof generation time and provide a similar breakdown for one-input one-output and two-input two-output.

	# gates
Witness validity	4204 (57.6%)
Delegated Schnorr	3059 (41.9%)
Asset equality	37 (0.5%)
Total	7300

Table 8: One-input-one-output.

	# gates
Witness validity	8408 (72.6%)
Delegated Schnorr	3059 (26.4%)
Asset equality	122 (1.1%)
Total	11589

Table 9: Two-input-two-output.

- For one-input one-output:
  - Proof generation in total takes 5.49 s on `m5.xlarge` and 3.51 s on `mac2.metal`, and in the case of multithreading, 3.73 s on `m5.xlarge` and 2.01 s on `mac2.metal`
  - The TurboPlonk proof takes 3.05 s on `m5.xlarge` and 1.89 s on `mac2.metal`, and in the case of multithreading, 1.91 s on `m5.xlarge` and 0.94 s on `mac2.metal`
  - The delegated Schnorr proof over `secp256k1` takes 0.005 s on `m5.xlarge` and 0.004 s on `mac2.metal`
  - Bulletproofs over `secq256k1` takes 2.30 s on `m5.xlarge` and 1.54 s on `mac2.metal`, and in the case of multithreading, 1.68 s on `m5.xlarge` and 0.97 s on `mac2.metal`
- For two-input two-output:
  - Proof generation in total takes 8.08 s on `m5.xlarge` and 5.36 s on `mac2.metal`, and in the case of multithreading, 5.50 s on `m5.xlarge` and 2.96 s on `mac2.metal`
  - The TurboPlonk proof takes 5.66 s on `m5.xlarge` and 3.74 s on `mac2.metal`, and in the case of multithreading, 3.68 s on `m5.xlarge` and 1.88 s on `mac2.metal`
  - The delegated Schnorr proof over `secq256k1` takes 0.005 s on `m5.xlarge` and 0.003 s on `mac2.metal`
  - Bulletproofs over `secq256k1` takes 2.28 s on `m5.xlarge` and 1.53 s on `mac2.metal`, and in the case of multithreading, 1.68 s on `m5.xlarge` and 0.99 s on `mac2.metal`

Similar to the result in Maxwell-Zerocash switching, the generation of TurboPlonk and Bulletproofs takes most of the time.

Proof size. The proof size is 2983 bytes (2.9 KB), which consists of:

- a delegated Schnorr proof over `secq256k1` of 419 bytes (14.0%)
- three Bulletproofs input commitments of 131 bytes (4.4%)
- a Bulletproofs over `secq256k1` of 1273 bytes (42.7%)
- a TurboPlonk proof of 1160 bytes (38.9%)

Note that the proof size for both cases is the same—because in our setup, the TurboPlonk proof has a constant size, and both the delegated Schnorr proof and Bulletproofs over `secq256k1` are checking only one proof of knowledge of the secret key, which remains unchanged.

Proof verification time. Similarly, we measure the proof verification time as well as its breakdown in the single-thread setting. Our observation is similar to that in the Maxwell-Zerocash switching.

- For one-input one-output:
  - Proof verification in total takes 0.239 s on `m5.xlarge` and 0.159 s on `mac2.metal`
  - Verification of the TurboPlonk proof takes 0.013 s on `m5.xlarge` and 0.009 s on `mac2.metal`
  - Verification of the delegated Schnorr proof over `secp256k1` takes 0.001 s on `m5.xlarge` and 0.001 s on `mac2.metal`
  - Verification of the Bulletproofs over `secq256k1` takes 0.225 s on `m5.xlarge` and 0.149 s on `mac2.metal`
- For two-input two-output:
  - Proof verification in total takes 0.239 s on `m5.xlarge` and 0.159 s on `mac2.metal`
  - Verification of the TurboPlonk proof takes 0.013 s on `m5.xlarge` and 0.009 s on `mac2.metal`
  - Verification of the delegated Schnorr proof over `secq256k1` takes 0.001 s on `m5.xlarge` and 0.001 s on `mac2.metal`
  - Verification of the Bulletproofs over `secq256k1` takes 0.225 s on `m5.xlarge` and 0.149 s on `mac2.metal`

6 Open problems

We are very interested in learning more about this area. Below is a list of open problems that we are interested in. Readers who would like to work on these problems can apply for a Findora Foundation research grant, which should be able to cover some miscellaneous costs. To start a discussion, email Findora Cryptography Team crypto@findora.org.

Discrete log equality:

1. In the Cryptography Stack Exchange, Geoffroy Couteau has noted that integer commitments (e.g., from RSA [DF02]) and range proof (e.g., for non-negativity [Lip03]) may be able to solve this problem, built over a construction by Yehuda Lindell in the same thread [Crya]. What does a concrete construction look like? What is its performance given recent results?
2. Is there a different way to modify the Chaum-Pedersen protocol, ideally using simple tools and without invoking general-purpose NIZK, that can achieve the same functionality (for discrete log equality, only) and is concretely efficient?

   This problem is solved by Melissa Chase (Microsoft Research), Michele Orrú (UC Berkeley), Trevor Perrin (Signal Foundation), and Greg Zaverucha (Microsoft Research) in their paper "Proofs of discrete logarithm equality across groups" [COPZ22].
3. Is there a way to prove the discrete log equality for groups with different orders, probably under some constraints, without using any nonnative field techniques?

   This problem is also solved in the paper "Proofs of discrete logarithm equality across groups" [COPZ22].

Inspection model:

1. Is there any existing work that actually has already covered an idea similar to inspection?

2. The inspection model in this paper is informal and imprecise. What would a formal treatment look like? What would be a better name for it that can put it in context with prior work?
3. We only have the delegated Schnorr protocol as an example of the inspection model. Are there more examples? Is the inspection model really useful for general protocol designs, or is it just a coincidence and is being lucky with the Schnorr protocol?
4. What is the connection between the inspection model and the accumulator-based proof-carrying data [BCMS20], which splits the verifier into an accumulation verifier and a decider? What is the connection of the inspection model with private aggregation by Boneh, Drake, Fisch, and Gabizon [BDFG21]?

Zerocash with secp256k1:

1. What are some other efficient constructions? Note that Bulletproofs is not the only proof system that is not subject to the needs of FFT space—we already list a few in Section 1. Note that we do not need to have "the best construction", as there is often a trade-off. For example, the use of TurboPlonk requires users to download the SRS, which in production is tiresome.
2. Most existing syntaxes for zero-knowledge proofs do not capture the existence of a transcript T, for example from another protocol. However, the transcript is heavily used in practice and across several protocols. Therefore, such syntax may not capture the actual implementation. What would the syntax look like for zero-knowledge proofs with explicit mutable access to a transcript (i.e., `&mut transcript`)? This is the situation when we use the Merlin Rust library [Mer] to manage the transcript.

Implementation:

- Note that we do not necessarily need to use `secq256k1`, and as we discuss in Section 1, there are some limitations of `secq256k1` though not relevant to our particular use case. Is there a better curve? Is there a general way, or a list of criteria, in finding a "good" curve?

Acknowledgment

The authors would like to thank Rami Akeela, David Galindo, Vipul Goyal, Yuncong Hu, Abhishek Jain, Jonathan Katz, Carla Ràfols, Tom Shrimpton, and Tiancheng Xie for discussion and feedback. The authors would like to especially thank Pratyush Mishra for pointing out `secq256k1` ("q").

A A binary-testing-friendly variant of TurboPlonk

We now describe the variant of TurboPlonk used in the Findora blockchain for the applications in this paper. Our construction follows the standard recipe of customizing TurboPlonk, so we do not claim any novelty. This variant has five wires, twelve selectors, high degrees, and additional polynomial identity constraints. It includes customized gates for Rescue and boolean testing. The

following equation summarizes the structure.

$$
\begin{aligned}
& q_1(X) \cdot w_1(X) + q_2(X) \cdot w_2(X) + q_3(X) \cdot w_3(X) + q_4(X) \cdot w_4(X) \quad \text{// linear combination} \\
& + q_{m1}(X) \cdot w_1(X) \cdot w_2(X) + q_{m2}(X) \cdot w_3(X) \cdot w_4(X) \quad \text{// multiplication (somewhat)} \\
& + q_c(X) \quad \text{// constants} \\
& + \mathsf{PI}(X) \quad \text{// inputs} \\
& + q_{hash1}(X) \cdot (w_1(X))^5 + q_{hash2}(X) \cdot (w_2(X))^5 \\
& + q_{hash3}(X) \cdot (w_3(X))^5 + q_{hash4}(X) \cdot (w_4(X))^5 \quad \text{// Rescue non-linear steps} \\
& = q_o(X) \cdot w_o(X) \quad \text{// output} \qquad (1)
\end{aligned}
$$

$$q_b(X) \cdot w_2(X) \cdot (w_2(X) - 1) = 0 \quad \text{// boolean testing on the second wire} \qquad (2)$$

$$q_b(X) \cdot w_3(X) \cdot (w_3(X) - 1) = 0 \quad \text{// boolean testing on the third wire} \qquad (3)$$

$$q_b(X) \cdot w_4(X) \cdot (w_4(X) - 1) = 0 \quad \text{// boolean testing on the fourth wire} \qquad (4)$$

We call this variant binary-testing-friendly since, in addition to the arithmetic condition (see eq. (1)), there are three boolean testing conditions (see eq. (2), eq. (3), and eq. (4)). In a non-binary-testing-friendly construction, these additional constraints would have been instantiated with additional gates, but one can apply them directly to an existing gate to restrict its 2nd, 3rd, and 4th input wires. This effectively reduces the cost of bit decomposition, an operation that uses a lot of binary testing, by 75%. Bit decomposition contributes to a lot of the cost in range checks, Merkle trees via Pedersen commitments, field simulation, and emulation of binary circuits.

A.1 Indexer

The indexer in TurboPlonk computes the commitments and openings for indexing polynomials, which consist of the following:

- The thirteen selector polynomials:

$$
\begin{aligned}
& q_1(X), q_2(X), q_3(X), q_4(X), q_o(X), q_{m1}(X), q_{m2}(X), q_c(X), \\
& q_{hash1}(X), q_{hash2}(X), q_{hash3}(X), q_{hash4}(X), q_b(X)
\end{aligned}
$$

- The five permutation polynomials:

$$S_{\sigma 1}(X), S_{\sigma 2}(X), S_{\sigma 3}(X), S_{\sigma 4}(X), S_{\sigma o}(X)$$

First of all, let the number of gates be n, the constraint system should have indicated a permutation $\sigma : [5n] \to [5n]$, which fulfills the following requirements.

- Let $w(X)$ be the concatenated witness polynomial of $w_1(X), w_2(X), w_3(X), w_4(X), w_o(X)$. The concatenation is over the evaluation representation, not the coefficient representation.
- The evaluation of $w(X)$ remains unchanged after applying σ as a permutation over the evaluation of $w(X)$ itself.

Now, given four independent quadratic non-residues (k_1, k_2, k_3, k_4) and a generator ω with order n, we define a mapping as follows.

$$\sigma_0(i) = \begin{cases} \omega^{i-1} & i \in \{1, 2, ..., n\} \\ k_1 \cdot \omega^{i-1-n} & i \in \{n+1, n+2, ..., 2n\} \\ k_2 \cdot \omega^{i-1-2n} & i \in \{2n+1, 2n+2, ..., 3n\} \\ k_3 \cdot \omega^{i-1-3n} & i \in \{3n+1, 3n+2, ..., 4n\} \\ k_4 \cdot \omega^{i-1-4n} & i \in \{4n+1, 4n+2, ..., 5n\} \end{cases}$$

where $i = 1, 2, ..., n$. And we apply this mapping to each element in σ, obtaining a map $\sigma^*(x) : [5n] \to \mathbb{F}$. We split this map into five permutation polynomials, as follows.

- $S_{\sigma 1}(X)$'s evaluation on $1, \omega, ..., \omega^{n-1}$ equals $\sigma^*(x)$'s evaluation on $1, 2, ..., n$.
- $S_{\sigma 2}(X)$'s evaluation on $1, \omega, ..., \omega^{n-1}$ equals $\sigma^*(x)$'s evaluation on $n+1, n+2, ..., 2n$.
- $S_{\sigma 3}(X)$'s evaluation on $1, \omega, ..., \omega^{n-1}$ equals $\sigma^*(x)$'s evaluation on $2n+1, 2n+2, ..., 3n$.
- $S_{\sigma 4}(X)$'s evaluation on $1, \omega, ..., \omega^{n-1}$ equals $\sigma^*(x)$'s evaluation on $3n+1, 3n+2, ..., 4n$.
- $S_{\sigma o}(X)$'s evaluation on $1, \omega, ..., \omega^{n-1}$ equals $\sigma^*(x)$'s evaluation on $4n+1, 4n+2, ..., 5n$.

Step 1: commit all polynomials. We first commit all these indexing polynomials. The commitments are included in the verifier parameters. We then perform some precomputation: we prepare a representation of these polynomials that are easy to use later for proving, by doing a coset FFT over them. The prepared polynomials are included in the prover parameters.

Step 2: precompute the two helper polynomials. Compute the following polynomial defined on a domain H of size n:

$$L_1(X) = \frac{X^n - 1}{X - 1}$$

and store its coset FFT representation. This is done by first observing that $L_1(X)$ evaluates to n on $X = 1$ and 0 otherwise in H. We can perform an inverse FFT to convert it back to the coefficient representation (which indeed looks nontrivial). Then, we perform a coset FFT, which gives us the prepared version of this polynomial.

Another polynomial we precompute is the vanishing polynomial of domain H of size n:

$$Z_H(X) = X^n - 1$$

and we want to store its coset FFT representation. This is done by a coset FFT over the coefficient representation above. The representations for the two helper polynomials are included in the prover parameters.

Step 3: compute the Lagrange interpolation constants. Recall that the Lagrange interpolation from $(1, y_0), (\omega, y_1), ..., (\omega^n, y_n)$ where ω is the generator for a domain H, into to a polynomial $f(X)$ of degree n is as follows:

$$f(X) = \sum_{j=0}^{n} \quad y_j \quad \left(\prod_{\substack{0 \le m \le n \\ m \ne j}} \quad \frac{X - \omega^m}{\omega^j - \omega^m} \right)$$

We can rewrite it as follows.

$$f(X) = \sum_{j=0}^{n} \quad y_j \quad \left(\prod_{\substack{0 \le m \le n \\ m \ne j}} \quad \frac{X - \omega^m}{\omega^j - \omega^m} \right)$$

$$= \left(\prod_{0 \le m \le n} (X - \omega^m) \right) \quad \left(\sum_{j=0}^{n} \quad \frac{y_j}{X - \omega^j} \quad \left(\prod_{\substack{0 \le m \le n \\ m \ne j}} \quad \frac{1}{\omega^j - \omega^m} \right) \right)$$

Now, precompute c_j for every $j \in \{0, 1, ..., n\}$.

$$c_j = \prod_{\substack{0 \le m \le n \\ m \ne j}} \quad \frac{1}{\omega^j - \omega^m}$$

This allows us to simplify $f(X)$ as follows.

$$f(X) = \left(\prod_{0 \le m \le n} (X - \omega^m) \right) \quad \sum_{j=0}^{n} \quad \frac{c_j \cdot y_j}{X - \omega^j}$$

$$= (X^n - 1) \quad \sum_{j=0}^{n} \quad \frac{c_j \cdot y_j}{X - \omega^j}$$

These constants c_j ($j \in \{0, 1, ..., n\}$) are included in the verifier parameters. We conclude the description of the indexer.

A.2 Prover

The prover in TurboPlonk uses the prover parameters from the indexer and a complete constraint system with all the gate values and copy constraints ready. It follows the following steps.

Step 1: assemble public inputs. The prover parameters have indicated which witness value indeed belongs to public inputs. The prover finds those witness values and stores them in a vector of length n_{in}, which is the number of field elements in public inputs. This is to enable us to calculate the state of the verifier.

Step 2: instantiate the verifier. For the purpose of the Fiat-Shamir transform, we create a cryptographic sponge, which will absorb the verifier's state as well as the messages that the verifier would receive from the prover in an interactive proof protocol.

After we create the sponge, we put the following two things into the sponge: (1) verifier parameters and (2) public inputs.

Step 3: commit witness polynomials with hiding. Given the witness polynomials $w_1(X)$, $w_2(X), w_3(X), w_4(X), w_o(X)$, we add a degree-one random blinding polynomial over each of them. The prover samples $b_1, b_2, b_3, ..., b_{10} \in \mathbb{F}$ and computes blinded witness polynomials.

$$\begin{aligned}
\widetilde{w_1}(X) &= w_1(X) + Z_H(X) \cdot (b_1 \cdot X + b_2) \\
\widetilde{w_2}(X) &= w_2(X) + Z_H(X) \cdot (b_3 \cdot X + b_4) \\
\widetilde{w_3}(X) &= w_3(X) + Z_H(X) \cdot (b_5 \cdot X + b_6) \\
\widetilde{w_4}(X) &= w_4(X) + Z_H(X) \cdot (b_7 \cdot X + b_8) \\
\widetilde{w_o}(X) &= w_o(X) + Z_H(X) \cdot (b_9 \cdot X + b_{10})
\end{aligned}$$

We commit each of the polynomials above and put the polynomial commitments $\mathsf{cm}_{w1}, \mathsf{cm}_{w2}$, $\mathsf{cm}_{w3}, \mathsf{cm}_{w4}, \mathsf{cm}_{wo} \in \mathbb{G}_1$ into the sponge.

Step 4: build the sigma polynomial, for wiring. The prover squeezes out two challenges $\beta, \gamma \in \mathbb{F}$ from the sponge. We now need to build the sigma polynomial. It helps for us to first compute:

$$S_i := \frac{\begin{array}{c}(w_i + \beta \cdot \omega^{i-1} + \gamma) \cdot (w_{n+i} + \beta \cdot k_1 \cdot \omega^{i-1} + \gamma) \cdot (w_{2n+i} + \beta \cdot k_2 \cdot \omega^{i-1} + \gamma) \\ \cdot (w_{3n+i} + \beta \cdot k_3 \cdot \omega^{i-1} + \gamma) \cdot (w_{4n+i} + \beta \cdot k_4 \cdot \omega^{i-1} + \gamma)\end{array}}{\begin{array}{c}(w_i + \sigma^*(i) \cdot \beta + \gamma) \cdot (w_{n+i} + \sigma^*(n+i) \cdot \beta + \gamma) \cdot (w_{2n+i} + \sigma^*(2n+i) \cdot \beta + \gamma) \\ \cdot (w_{3n+i} + \sigma^*(3n+i) \cdot \beta + \gamma) \cdot (w_{4n+i} + \sigma^*(4n+i) \cdot \beta + \gamma)\end{array}}$$

We can then define the permutation polynomial $z(X)$ with the following evaluations:

$$z(\omega^{i-1}) = \begin{cases} 1 & i = 1 \\ \prod_{j=1}^{i-1} S_i & i = 2, 3, ..., n \end{cases}$$

Step 5: commit the sigma polynomial, with hiding. The prover first samples $b_{11}, b_{12}, b_{13} \in \mathbb{F}$ and applies them as blinding factors to the polynomial $z(X)$.

$$\widetilde{z}(X) = z(X) + Z_H(X) \cdot (b_{11}X^2 + b_{12}X + b_{13})$$

We commit this polynomial and put the polynomial commitment $\mathsf{cm}_z \in \mathbb{G}_1$ into the sponge.

Step 6: compute the quotient polynomial. The prover squeezes out a challenge α from the sponge. This is used to construct the following polynomial.

$$\begin{aligned}
t(X) = t_{\mathsf{sat}}(X) \cdot \frac{1}{Z_H(X)} + t_{\sigma 1}(X) \cdot \frac{\alpha}{Z_H(X)} - t_{\sigma 2}(X) \cdot \frac{\alpha}{Z_H(X)} + t_{\sigma 3}(X) \cdot \frac{\alpha^2}{Z_H(X)} \\
+ t_{b1}(X) \cdot \frac{\alpha^3}{Z_H(X)} + t_{b2}(X) \cdot \frac{\alpha^4}{Z_H(X)} + t_{b3}(X) \cdot \frac{\alpha^5}{Z_H(X)}
\end{aligned}$$

where

$$\begin{aligned}
t_{sat}(X) = {} & q_1(X) \cdot \widetilde{w_1}(X) + q_2(X) \cdot \widetilde{w_2}(X) + q_3(X) \cdot \widetilde{w_3}(X) + q_4(X) \cdot \widetilde{w_4}(X) \\
& + q_{m1}(X) \cdot \widetilde{w_1}(X) \cdot \widetilde{w_2}(X) + q_{m2}(X) \cdot \widetilde{w_3}(X) \cdot \widetilde{w_4}(X) + q_c(X) + \mathsf{PI}(X) \\
& + q_{hash1}(X) \cdot (\widetilde{w_1}(X))^5 + q_{hash2}(X) \cdot (\widetilde{w_2}(X))^5 \\
& + q_{hash3}(X) \cdot (\widetilde{w_3}(X))^5 + q_{hash4}(X) \cdot (\widetilde{w_4}(X))^5 \\
& - q_o(X) \cdot \widetilde{w_o}(X)
\end{aligned}$$

and

$$\begin{aligned} t_{\sigma 1}(X) =& (\widetilde{w_1}(X) + \beta \cdot X + \gamma) \cdot (\widetilde{w_2}(X) + \beta \cdot k_1 \cdot X + \gamma) \cdot (\widetilde{w_3}(X) + \beta \cdot k_2 \cdot X + \gamma) \\ & \cdot (\widetilde{w_4}(X) + \beta \cdot k_3 \cdot X + \gamma) \cdot (\widetilde{w_o}(X) + \beta \cdot k_4 \cdot X + \gamma) \\ & \cdot \widetilde{z}(X) \end{aligned}$$

and

$$\begin{aligned} t_{\sigma 2}(X) =& (\widetilde{w_1}(X) + \beta \cdot S_{\sigma 1}(X) + \gamma) \cdot (\widetilde{w_2}(X) + \beta \cdot S_{\sigma 2}(X) + \gamma) \cdot (\widetilde{w_3}(X) + \beta \cdot S_{\sigma 3}(X) + \gamma) \\ & \cdot (\widetilde{w_4}(X) + \beta \cdot S_{\sigma 4}(X) + \gamma) \cdot (\widetilde{w_o}(X) + \beta \cdot S_{\sigma o}(X) + \gamma) \\ & \cdot \widetilde{z}(X\omega) \quad \text{// the shifting trick} \end{aligned}$$

and

$$t_{\sigma 3}(X) = (\widetilde{z}(X) - 1) \cdot L_1(X)$$

and

$$\begin{aligned} t_{b1}(X) &= q_b(X) \cdot \widetilde{w_2}(X) \cdot (1 - \widetilde{w_2}(X)) \\ t_{b2}(X) &= q_b(X) \cdot \widetilde{w_3}(X) \cdot (1 - \widetilde{w_3}(X)) \\ t_{b3}(X) &= q_b(X) \cdot \widetilde{w_4}(X) \cdot (1 - \widetilde{w_4}(X)) \end{aligned}$$

Then, in the coefficient representations, we split the polynomial into five parts: $t_1(X)$, $t_2(X)$, $t_3(X)$, $t_4(X)$, and $t_5(X)$, where the first four polynomials have degree $n+2$, and the last polynomial has degree $n-1$. This is because $t(X)$ is expected to have degree $5 \cdot (n+1) + (n+2) - n = 5n + 7$, and $5n + 7 = 4 \cdot (n+2) + (n-1)$.

Step 7: commit the split quotient polynomials, without hiding. We commit all these polynomials and put the commitments cm_{t1}, cm_{t2}, cm_{t3}, cm_{t4}, and cm_{t5} into the sponge.

Step 8: open the polynomials at a random point. The prover squeezes a random challenge $\zeta \in \mathbb{F}$ and computes the following opening evaluations:

$$\widetilde{w_1}(\zeta), \widetilde{w_2}(\zeta), \widetilde{w_3}(\zeta), \widetilde{w_4}(\zeta), \widetilde{w_o}(\zeta), S_{\sigma 1}(\zeta), S_{\sigma 2}(\zeta), S_{\sigma 3}(\zeta), S_{\sigma 4}(\zeta), \widetilde{z}(\zeta\omega)$$

And we remind the readers that the last one is evaluated over $\zeta\omega$ instead of ζ. This is common in entry product arguments. The prover puts these, which are elements in $\mathbb{F}$, into the sponge.

Step 9: compute the linearization polynomial. The linearization polynomial is, at a high level, to replace $\widetilde{w_1}(X)$, $\widetilde{w_2}(X)$, $\widetilde{w_3}(X)$, $\widetilde{w_4}(X)$, $\widetilde{w_o}(X)$, $\widetilde{S_{\sigma 1}}(X)$, $\widetilde{S_{\sigma 2}}(X)$, $\widetilde{S_{\sigma 3}}(X)$, $\widetilde{S_{\sigma 4}}(X)$, and $\widetilde{z}(X\omega)$ with the evaluations over that random point. More specifically, $r(X)$ reads as follows. Note that the locations of $Z_H(X)$ are different now. The constant terms of $r(X)$ are removed here.

$$\begin{aligned} r(X) = & r_{\mathsf{sat}}(X) + r_1(X) \cdot \alpha - r_2(X) \cdot \alpha + r_3(X) \cdot \alpha^2 + r_4(X) \cdot \alpha^3 + r_5(X) \cdot \alpha^4 + r_6(X) \cdot \alpha^5 \\ & - Z_H(\zeta)(t_1(X) + \zeta^{n+2} \cdot t_2(X) + \zeta^{2(n+2)} \cdot t_3(X) + \zeta^{3(n+2)} \cdot t_4(X) + \zeta^{4(n+2)} \cdot t_5(X)) \end{aligned}$$

where

$$\begin{aligned} r_{\mathsf{sat}}(X) = & \widetilde{w_1}(\zeta) \cdot q_1(X) + \widetilde{w_2}(\zeta) \cdot q_2(X) + \widetilde{w_3}(\zeta) \cdot q_3(X) + \widetilde{w_4}(\zeta) \cdot q_4(X) \\ & + \widetilde{w_1}(\zeta) \cdot \widetilde{w_2}(\zeta) \cdot q_{m1}(X) + \widetilde{w_3}(\zeta) \cdot \widetilde{w_4}(\zeta) \cdot q_{m2}(X) + q_c(X) \\ & + (\widetilde{w_1}(\zeta))^5 \cdot q_{hash1}(X) + (\widetilde{w_2}(\zeta))^5 \cdot q_{hash2}(X) \\ & + (\widetilde{w_3}(\zeta))^5 \cdot q_{hash3}(X) + (\widetilde{w_4}(\zeta))^5 \cdot q_{hash4}(X) \\ & - \widetilde{w_o}(\zeta) \cdot q_o(X) \end{aligned}$$

and

$$\begin{aligned} r_1(X) = & (\widetilde{w_1}(\zeta) + \beta \cdot \zeta + \gamma) \cdot (\widetilde{w_2}(\zeta) + \beta \cdot k_1 \cdot \zeta + \gamma) \cdot (\widetilde{w_3}(\zeta) + \beta \cdot k_2 \cdot \zeta + \gamma) \\ & \cdot (\widetilde{w_4}(\zeta) + \beta \cdot k_3 \cdot \zeta + \gamma) \cdot (\widetilde{w_o}(\zeta) + \beta \cdot k_4 \cdot \zeta + \gamma) \cdot \widetilde{z}(X) \end{aligned}$$

and

$$\begin{aligned} r_2(X) = & (\widetilde{w_1}(\zeta) + \beta S_{\sigma 1}(\zeta) + \gamma) \cdot (\widetilde{w_2}(\zeta) + \beta S_{\sigma 2}(\zeta) + \gamma) \cdot (\widetilde{w_3}(\zeta) + \beta S_{\sigma 3}(\zeta) + \gamma) \\ & \cdot (\widetilde{w_4}(\zeta) + \beta S_{\sigma 4}(\zeta) + \gamma) \cdot \beta \cdot S_{\sigma o}(X) \cdot \widetilde{z}(\zeta\omega) \end{aligned}$$

Note that $S_{\sigma o}(X)$ is not being replaced by its evaluation, and

$$r_3(X) = \widetilde{z}(X) \cdot L_1(\zeta)$$

and

$$\begin{aligned} r_4(X) &= q_b(X) \cdot \widetilde{w_2}(\zeta) \cdot (\widetilde{w_2}(\zeta) - 1) \\ r_5(X) &= q_b(X) \cdot \widetilde{w_3}(\zeta) \cdot (\widetilde{w_3}(\zeta) - 1) \\ r_6(X) &= q_b(X) \cdot \widetilde{w_4}(\zeta) \cdot (\widetilde{w_4}(\zeta) - 1) \end{aligned}$$

Now we have the linearization polynomial. Although this polynomial is not linear, its degree is down from $\approx 5n$ to $\approx n$, and one can see that the polynomial collapses to a shorter one.

We note that the evaluation of $r(X)$ at point ζ is as follows. The prover does not need to include this number in the proof because it can be computed by the verifier.

$$\begin{aligned} r(\zeta) = & -\mathsf{PI}(\zeta) + \alpha \cdot (\widetilde{w_1}(\zeta) + \beta S_{\sigma 1}(\zeta) + \gamma) \cdot (\widetilde{w_2}(\zeta) + \beta S_{\sigma 2}(\zeta) + \gamma) \cdot (\widetilde{w_3}(\zeta) + \beta S_{\sigma 3}(\zeta) + \gamma) \\ & \cdot (\widetilde{w_4}(\zeta) + \beta S_{\sigma 4}(\zeta) + \gamma) \cdot (\widetilde{w_o}(\zeta) + \gamma) \cdot \widetilde{z}(\zeta\omega) + \alpha^2 L_1(\zeta) \end{aligned}$$

Step 10: compute the opening proof polynomials. Now we want to prove that the previous openings are correct, as well as $r(X)$ is actually vanishing in H. We first squeeze out a challenge

$v \in \mathbb{F}$ from the sponge. This is done by showing that one can find a polynomial $W_\zeta(X)$ such that:

$$W_\zeta(X) = \frac{1}{X-\zeta}\begin{pmatrix} \widetilde{w_1}(X) - \widetilde{w_1}(\zeta) \\ +v(\widetilde{w_2}(X) - \widetilde{w_2}(\zeta)) \\ +v^2(\widetilde{w_3}(X) - \widetilde{w_3}(\zeta)) \\ +v^3(\widetilde{w_4}(X) - \widetilde{w_4}(\zeta)) \\ +v^4(\widetilde{w_o}(X) - \widetilde{w_o}(\zeta)) \\ +v^5(S_{\sigma 1}(X) - S_{\sigma 1}(\zeta)) \\ +v^6(S_{\sigma 2}(X) - S_{\sigma 2}(\zeta)) \\ +v^7(S_{\sigma 3}(X) - S_{\sigma 3}(\zeta)) \\ +v^8(S_{\sigma 4}(X) - S_{\sigma 4}(\zeta)) \\ +v^9(r(X) - r(\zeta)) \end{pmatrix}$$

and similar another polynomial $W_{\zeta\omega}(X)$ as follows.

$$W_{\zeta\omega}(X) = \frac{\widetilde{z}(X) - \widetilde{z}(\zeta\omega)}{X - \zeta\omega}$$

We commit these two polynomials as cm_ζ and $\mathsf{cm}_{\zeta\omega}$.

Step 11: output the full proof. After the Fiat-Shamir transform, the final proof reads as follows.

$$\begin{pmatrix} \mathsf{cm}_{w1}, \mathsf{cm}_{w2}, \mathsf{cm}_{w3}, \mathsf{cm}_{w4}, \mathsf{cm}_{wo}, \\ \mathsf{cm}_z, \\ \mathsf{cm}_{t1}, \mathsf{cm}_{t2}, \mathsf{cm}_{t3}, \mathsf{cm}_{t4}, \mathsf{cm}_{t5}, \\ \widetilde{w_1}(\zeta), \widetilde{w_2}(\zeta), \widetilde{w_3}(\zeta), \widetilde{w_4}(\zeta), \widetilde{w_o}(\zeta), \\ S_{\sigma 1}(\zeta), S_{\sigma 2}(\zeta), S_{\sigma 3}(\zeta), S_{\sigma 4}(\zeta), \\ \widetilde{z}(\zeta\omega), \\ \mathsf{cm}_\zeta, \mathsf{cm}_{\zeta\omega} \end{pmatrix}$$

A.3 Verifier

The verifier reads the full proof above and proceeds as follows.

Step 1: compute all the challenges. For convenience, the verifier first computes all the random challenges in this protocol execution: β, γ, α, ζ, v, and u.

- Initialize the same cryptographic sponge.
- Put $\mathsf{cm}_{w1}, \mathsf{cm}_{w2}, \mathsf{cm}_{w3}, \mathsf{cm}_{w4}, \mathsf{cm}_{wo}$ into the sponge and squeeze out $\beta, \gamma \in \mathbb{F}$ from the sponge.
- Put cm_z into the sponge and squeeze out $\alpha \in \mathbb{F}$ from the sponge.
- Put $\mathsf{cm}_{t1}, \mathsf{cm}_{t2}, \mathsf{cm}_{t3}, \mathsf{cm}_{t4}, \mathsf{cm}_{t5}$ into the sponge and squeeze out $\zeta \in \mathbb{F}$ from the sponge.
- Put $\widetilde{w_1}(\zeta), \widetilde{w_2}(\zeta), \widetilde{w_3}(\zeta), \widetilde{w_4}(\zeta), \widetilde{w_o}(\zeta), S_{\sigma 1}(\zeta), S_{\sigma 2}(\zeta), S_{\sigma 3}(\zeta), S_{\sigma 4}(\zeta), \widetilde{z}(\zeta\omega)$ into the sponge and squeeze out $v \in \mathbb{F}$ from the sponge.
- Put cm_ζ and $\mathsf{cm}_{\zeta\omega}$ into the sponge and squeeze out $u \in \mathbb{F}$ from the sponge.

Step 2: compute $r(\zeta)$. Recall from the Step 9 of the prover, $r(\zeta)$ can indeed be computed by the verifier.

$$\begin{aligned} r(\zeta) = &-\mathsf{PI}(\zeta) + \alpha \cdot (\widetilde{w_1}(\zeta) + \beta S_{\sigma 1}(\zeta) + \gamma) \cdot (\widetilde{w_2}(\zeta) + \beta S_{\sigma 2}(\zeta) + \gamma) \cdot (\widetilde{w_3}(\zeta) + \beta S_{\sigma 3}(\zeta) + \gamma) \\ &\cdot (\widetilde{w_4}(\zeta) + \beta S_{\sigma 4}(\zeta) + \gamma) \cdot (\widetilde{w_o}(\zeta) + \gamma) \cdot \widetilde{z}(\zeta\omega) + \alpha^2 L_1(\zeta) \end{aligned}$$

Step 3: assemble cm_r. The verifier can also assemble the commitment of $r(X)$ from available commitments, as follows.

$$\begin{aligned} \mathsf{cm}_r = &\ \mathsf{cm}_{sat} + \mathsf{cm}_{r1} \cdot \alpha - \mathsf{cm}_{r2} \cdot \alpha + \mathsf{cm}_{r3} \cdot \alpha^2 \\ &- Z_H(\zeta) \cdot (\mathsf{cm}_{t1} + \zeta^{n+2} \cdot \mathsf{cm}_{t2} + \zeta^{2(n+2)} \cdot \mathsf{cm}_{t3} + \zeta^{3(n+2)} \cdot \mathsf{cm}_{t4} + \zeta^{4(n+2)} \cdot \mathsf{cm}_{t5}) \\ &+ \mathsf{cm}_b \cdot (\widetilde{w_2}(\zeta) \cdot (\widetilde{w_2}(\zeta) - 1) \cdot \alpha^3 + \widetilde{w_3}(\zeta) \cdot (\widetilde{w_3}(\zeta) - 1) \cdot \alpha^4 + \widetilde{w_4}(\zeta) \cdot (\widetilde{w_4}(\zeta) - 1) \cdot \alpha^5) \end{aligned}$$

where

$$\begin{aligned} \mathsf{cm}_{sat} = &\ \widetilde{w_1}(\zeta) \cdot \mathsf{cm}_{q1} + \widetilde{w_2}(\zeta) \cdot \mathsf{cm}_{q2} + \widetilde{w_3}(\zeta) \cdot \mathsf{cm}_{q3} + \widetilde{w_4}(\zeta) \cdot \mathsf{cm}_{q4} \\ &+ \widetilde{w_1}(\zeta) \cdot \widetilde{w_2}(\zeta) \cdot \mathsf{cm}_{m1} + \widetilde{w_3}(\zeta) \cdot \widetilde{w_4}(\zeta) \cdot \mathsf{cm}_{m2} + \mathsf{cm}_{qc} \\ &+ \widetilde{w_1}(\zeta) \cdot \widetilde{w_2}(\zeta) \cdot \widetilde{w_3}(\zeta) \cdot \widetilde{w_4}(\zeta) \cdot \widetilde{w_o}(\zeta) \cdot \mathsf{cm}_{ecc} \\ &+ (\widetilde{w_1}(\zeta))^5 \cdot \mathsf{cm}_{hash1} + (\widetilde{w_2}(\zeta))^5 \cdot \mathsf{cm}_{hash2} \\ &+ (\widetilde{w_3}(\zeta))^5 \cdot \mathsf{cm}_{hash3} + (\widetilde{w_4}(\zeta))^5 \cdot \mathsf{cm}_{hash4} \\ &- \widetilde{w_o}(\zeta) \cdot \mathsf{cm}_o \end{aligned}$$

and

$$\begin{aligned} \mathsf{cm}_{r1} = &(\widetilde{w_1}(\zeta) + \beta \cdot \zeta + \gamma) \cdot (\widetilde{w_2}(\zeta) + \beta \cdot k_1 \cdot \zeta + \gamma) \cdot (\widetilde{w_3}(\zeta) + \beta \cdot k_2 \cdot \zeta + \gamma) \\ &\cdot (\widetilde{w_4}(\zeta) + \beta \cdot k_3 \cdot \zeta + \gamma) \cdot (\widetilde{w_o}(\zeta) + \beta \cdot k_4 \cdot \zeta + \gamma) \cdot \mathsf{cm}_z \end{aligned}$$

and

$$\begin{aligned} \mathsf{cm}_{r2} = &(\widetilde{w_1}(\zeta) + \beta S_{\sigma 1}(\zeta) + \gamma) \cdot (\widetilde{w_2}(\zeta) + \beta S_{\sigma 2}(\zeta) + \gamma) \cdot (\widetilde{w_3}(\zeta) + \beta S_{\sigma 3}(\zeta) + \gamma) \\ &\cdot (\widetilde{w_4}(\zeta) + \beta S_{\sigma 4}(\zeta) + \gamma) \cdot \beta \cdot \widetilde{z}(\zeta\omega) \cdot \mathsf{cm}_{\sigma o} \end{aligned}$$

and

$$\mathsf{cm}_{r3} = L_1(\zeta) \cdot \mathsf{cm}_z$$

Step 4: compute linear combination of commitments. Let the cumulative commitment cm be as follows. Note that the last one has a coefficient u.

$$\begin{aligned} \mathsf{cm} = &\ \mathsf{cm}_{w1} + v \cdot \mathsf{cm}_{w2} + v^2 \cdot \mathsf{cm}_{w3} + v^3 \cdot \mathsf{cm}_{w4} + v^4 \cdot \mathsf{cm}_{wo} \\ &+ v^5 \cdot \mathsf{cm}_{\sigma 1} + v^6 \cdot \mathsf{cm}_{\sigma 2} + v^7 \cdot \mathsf{cm}_{\sigma 3} + v^8 \cdot \mathsf{cm}_{\sigma 4} + v^9 \cdot \mathsf{cm}_r + u \cdot \mathsf{cm}_z \end{aligned}$$

Step 5: compute linear combination of evaluations. Let the cumulative evaluation s be as follows. Also, note the last one.

$$\begin{aligned} s = & \widetilde{w_1}(\zeta) + v \cdot \widetilde{w_2}(\zeta) + v^2 \cdot \widetilde{w_3}(\zeta) + v^3 \cdot \widetilde{w_4}(\zeta) + v^4 \cdot \widetilde{w_o}(\zeta) \\ & + v^5 \cdot S_{\sigma 1}(\zeta) + v^6 \cdot S_{\sigma 2}(\zeta) + v^7 \cdot S_{\sigma 3}(\zeta) + v^8 \cdot S_{\sigma 4}(\zeta) + v^9 \cdot r(\zeta) + u \cdot \widetilde{z}(\zeta\omega) \end{aligned}$$

Step 6: pairing. Compute L, R as follows.

$$L = e((\mathsf{cm}_\zeta + u \cdot \mathsf{cm}_{\zeta\omega}), x \cdot H)$$

where H is the generator of $\mathbb{G}_2$ in the SRS and x is the secret in the SRS. The element $x \cdot H$ here is part of the SRS.

$$R = e((\zeta \cdot \mathsf{cm}_\zeta + u \cdot \zeta \cdot \omega \cdot \mathsf{cm}_{\zeta\omega} + \mathsf{cm} - s \cdot G), H)$$

where G is the generator of $\mathbb{G}_1$ in the SRS.

Step 7: decision. The verifier accepts the proof if $L = R$ and rejects otherwise.

B (Im)possibility of fully MetaMask-compatible Zerocash

At the end of this paper we want to further the discussion of making the user experience of Zerocash "close to" that of EVM chains such as Ethereum, BSC, and Polygon. The transition from special curves like Jubjub or Grumpkin to `secp256k1` for the public key is just the first step. This allows the user to use the same key in different chains. Particularly, if you know another user's address, you can send tokens anonymously to this user on a different chain that supports the Zerocash construction (very few), while the other user does not need to register on that chain.

However, this is still not the same experience. An important reason for many chains to be aligned with Ethereum, for example using `secp256k1` for signatures and `0x` addresses, is that users can use MetaMask (or other similar browser-extension wallets) to interact with the network, including sending and receiving cryptoassets, invoking smart contracts, and swapping between cryptoassets. The most important functionality of MetaMask is probably to enable interactions with third-party dApps without giving away the user's private key, without which it would be difficult to imagine how a user can stake tokens or provide liquidity on the browser.

Chains not MetaMask-compatible. Chains that are not MetaMask-compatible, often due to the use of a different address system as in the case of Solana, would need to have their own browser extensions. Solana, for example, has been maintaining a list of wallet projects that support Solana, though they are clearly not as popular as MetaMask [Sol]. This gives rise to a lot of issues:

- Users who are already using MetaMask need to install yet another, less popular, wallet browser extension. This kind of the app often competes with each other when a dApp makes a request and can affect the user experience significantly.
- Users may simply decline to install another browser extension due to security concerns.

- The different address system may already prevent it from being compatible with the enormous ecosystem of existing dApps that surround the EVM.

These issues also affect any Zerocash construction that uses special curves for signatures. In this paper, we first avoid this problem by returning to `secp256k1`. However, it is still insufficient to become MetaMask-compatible, as we now discuss.

Impossibility: a standard nullifier. We first discuss why a standard nullifier, which is required for the Zerocash construction, is impossible. In the Zerocash construction, to send a Zerocash asset, one needs to compute its nullifier, which requires something related to the sender's secret key. In MetaMask, a third-party dApp cannot obtain the secret key (this is by design), and the closest we can get is MetaMask's signing API, which is a deterministic ECDSA signing oracle, following the deterministic ECDSA signature algorithm in RFC 6979 [Rfc].

The deterministic ECDSA signature gives us some hope—we can obtain the same signature for the same message, and this deterministic signature could probably serve as the nullifier. The problem is that MetaMask does not give us a proof that this signature follows exactly RFC 6979, and we now want to explain why this rules out using the deterministic signature as the nullifier.

RFC 6979 computes the randomizer in ECDSA by hashing the message as well as the secret key. To prove that a randomizer is generated in this way, we need to show that it is the result of such hashing. However, in the random oracle model, the only way to do so is to know the preimage, i.e., the message and the secret key, which is not the case here.

Unfortunately, the deterministic ECDSA signature gives us false hope—a standard nullifier as in the Zerocash construction is impossible at this moment.

Possibility: a holed nullifier. Although a standard nullifier cannot be made, there can be something similar to a nullifier, but with some caveats. Recall that a standard nullifier needs a secret key so that the previous sender (who creates the commitment and knows the public key) cannot predict the nullifier. If we allow the previous sender to predict the nullifier, which would enable the previous sender to "trace" the asset for one more step, then a nullifier that does not use a secret key can also be made. Recall that a commitment in the Zerocash construction is a hash commitment that can be represented as follows:

$$\mathsf{comm} := H(\mathsf{owner\text{-}address} \parallel \mathsf{amount} \parallel \mathsf{asset\text{-}type} \parallel \mathsf{randomizer})$$

We now consider another independently sampled hash function H' (which can be just H but with a different domain separator) and define what we call "holed nullifier" as follows:

$$\mathsf{nullifier} := H'(\mathsf{owner\text{-}address} \parallel \mathsf{amount} \parallel \mathsf{asset\text{-}type} \parallel \mathsf{randomizer})$$

Note that although the previous sender can predict this nullifier, the previous sender cannot do more than that. For example, the previous sender cannot spend this transaction on behalf of the user because the Zerocash construction checks the knowledge of the secret key.

We call this "holed nullifier" because it is very close to being a nullifier, except with such a linkability. Readers may feel that this holed nullifier cannot be used in production, but this is not

actually the case—if the previous sender is the user himself/herself, then the hole is not an issue. Later, we will further discuss this special case.

Impossibility: a standard owner's memo. In the Zerocash construction, the sender will encrypt the transaction information under the recipient's key and, for data availability, leave this ciphertext on chain, which we call the owner's memo. The recipient can find this ciphertext from the blockchain and obtain the transaction information through decryption. For example, generating either the standard or the holed nullifier requires the knowledge of the randomizer, which is included in the owner's memo. There are two challenges.

The first challenge has less to do with MetaMask but more with the Ethereum-style ECDSA signature verification. Note that in Ethereum, addresses are different from public keys. Each address is a hashed version of the public key. Ethereum can still do the ECDSA signature verification against this address because it can perform the signature verification in the reverse direction, known as recovering the public key from the signature (as in the precompiled contract "`ecrecover`"), and one can check the signature by hashing this public key and comparing against the address.

Although this reverse direction trick works for signature verification, it does not work with public key encryption. If a sender does not know the recipient's public key (which is behind the address), the sender cannot create this ciphertext. Generally, people can use tools to do a reverse lookup for the corresponding public key on the blockchain, if the address has ever made transactions. This does not work for completely new addresses.

There is no easy solution unless we assume that the reverse lookup always works. This is already restricted. Another problem is that reverse lookup leaks the recipient's address to the servers that provide this lookup service. Although this seems to be a problem solvable with private information retrieval (PIR), we do not have such an infrastructure yet, and we do not know if it is practical.

The second challenge is about how a recipient can decrypt the memo. MetaMask does not provide an interface to use the exact ECDSA secret key for decryption. First, readers may have concerns whether using the same key for signature and encryption is secure. This actually has been proven in the generic group model (GGM) [DLPSS12], if the encryption follows ECIES (also known as the hybrid encryption based on Diffie-Hellman key exchange). Nevertheless, MetaMask does not provide such a decryption oracle. The closest we can find within MetaMask is "`eth_getEncryptionPublicKey`", which is deprecated, that allows a user to have a separate pair of encryption keys derived from the signing key, but the user needs to publish this encryption key, which leads to additional overhead and privacy leakage.

In sum, it is unlikely that we can delegate the secret key of the Zerocash assets to MetaMask due to the owner's memo. We now describe a possible workaround.

Possibility: no owner's memo. It is important to note that, if a transaction in the Zerocash construction does not have the owner's memo, the recipient may still be able to spend the transaction, if the sender tells the recipient such transaction information that should have been included in the owner's memo. There are many out-of-band communication channels for the sender to do so, with different security and privacy tradeoffs. Readers may feel that this is very restricted and cannot be used in production because of its complexity. However, this is not always the case. Think about

the case if the sender and the recipient are indeed the same person, then there is no need to even exchange the owner's memo. We now discuss this idea further.

The road ahead: a Zerocash checking account. There appears to be a very narrow path toward MetaMask-compatible Zerocash, which avoids the need for a standard nullifier and an owner's memo by having the sender and the recipient be the same person. We want to note that this is indeed a very common setting, as we now discuss.

Assume that a user uses the Zerocash construction just for a secret checking account. When the user obtains tokens from elsewhere, the user always converts them into the Zerocash construction and puts them into this checking account. When the user wants to spend some tokens, the user gets the tokens from the checking account. Here, there are only two kinds of the transfer that the user needs to perform:

- Given the input of (1) the old checking account UTXO in the Zerocash construction and (2) a new, transparent asset UTXO, assuming that both are of the same asset type, the user can create a transaction that merges these two inputs together and produces an output, which is the new checking account UTXO, also in the Zerocash construction.
- Given the input of the old checking account UTXO in the Zerocash construction, the user can move some tokens out and let them become a new, transparent asset UTXO, while the rest of the tokens stay in the checking account. The output consists of (1) a new, transparent asset UTXO and (2) the new checking account UTXO in the Zerocash construction, which are of the same asset type.

This setting is indeed common for a chain that bridges together an UTXO chain and an EVM chain, where many transparent assets come from the EVM chain, probably from a cross-chain bridge, and the UTXO chain becomes a ledger for such Zerocash checking accounts. We already see one or two blockchain systems adopting this design.

References

[AABSDS20] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. "Design of symmetric-key primitives for advanced cryptographic protocols". In: *TOSC '20*.

[AGRRT16] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. "MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity". In: *ASIACRYPT '16*.

[Alb+19] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. "Feistel structures for MPC, and more". In: *ESORICS '19*.

[Azt] *Grumpkin addresses*. URL: `https://github.com/AztecProtocol/docs/blob/main/docs/sdk/types/barretenberg/GrumpkinAddress.md`.

[BBBPWM18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. "Bulletproofs: Short proofs for confidential transactions and more". In: *S&P '18*.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. "Gemini: Elastic SNARKs for diverse environments". In: *EUROCRYPT '22*.

[BCMS20] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nick Spooner. "Proof-carrying data from accumulation schemes". In: *TCC '20*.

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. "Halo Infinite: Proof-carrying data from additive polynomial commitments". In: *CRYPTO '21*.

[BG81] Charles H. Bennett and John Gill. "Relative to a random oracle A, $P^A \neq NP^A \neq co\text{-}NP^A$ with probability 1". In: *SICOMP '81*.

[BS07] Reinier Broker and Peter Stevenhagen. "Constructing elliptic curves of prime order". In: *Mathematics of Computation '07*.

[BS+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. "Zerocash: Decentralized anonymous payments from Bitcoin". In: *S&P '14*.

[BSBHR18] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. "Scalable, transparent, and post-quantum secure computational integrity". In: *IACR ePrint' 18*.

[BSCKL21] Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. "Elliptic curve fast Fourier transform (ECFFT) part I: Fast polynomial algorithms over all finite fields". In: *ECCC '21*.

[BSCKL22] Eli Ben-Sasson, Dan Carmon, Swastik Kopparty, and David Levit. "Scalable and transparent proofs over all large fields, via elliptic curves (ECFFT part II)". In: *ECCC '22*.

[Bit+13] Nir Bitansky, Dana Dachman-Soled, Sanjam Garg, Abhishek Jain, Yael Tauman Kalai, Adriana López-Alt, and Daniel Wichs. "Why "Fiat-Shamir for proofs" lacks a proof". In: *TCC '13*.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. "The random oracle methodology, revisited". In: *STOC '98*.

[CHMMVW20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas P. Ward. "Marlin: Preprocessing zkSNARKs with universal and updatable SRS". In: *EUROCRYPT '20*.

[CL20] Alessandro Chiesa and Siqi Liu. "On the impossibility of probabilistic proofs in relativized worlds". In: *ITCS '20*.

[CMT12] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. "Practical verified computation with streaming interactive proofs". In: *ITCS '12*.

[COPZ22] Melissa Chase, Michele Orrú, Trevor Perrin, and Greg Zaverucha. "Proofs of discrete logarithm equality across groups". In: *IACR ePrint 2022/1593*.

[CP92] David Chaum and Torben Pryds Pedersen. "Wallet databases with observers". In: *CRYPTO '92*.

[Cai] *Cairo*. URL: https://www.cairo-lang.org/.

[Coi] *Anonymous user sends ETH from Tornado Cash to prominent figures following sanctions*. URL: https://cointelegraph.com/news/anonymous-user-sends-eth-from-tornado-cash-to-prominent-figures-following-sanctions.

[Crya] *Discrete logarithm equality for independent groups*. https://crypto.stackexchange.com/questions/44744/discrete-logarithm-equality-for-independent-groups.

[Cryb] *Tornado Cash user continues to send ETH to prominent figures, Buterin admits he used the mixer*. URL: https://cryptonews.com/news/tornado-cash-user-continues-to-send-eth-to-prominent-figures-buterin-admits-he-used-the-mixer.htm.

[DF02] Ivan Damgård and Eiichiro Fujisaki. "A statistically-hiding integer commitment scheme based on groups with hidden order". In: *ASIACRYPT '02*.

[DLPSS12] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefler. "On the joint security of encryption and signature in EMV". In: *CT-RSA '12*.

[Dala] *Bulletproofs*. URL: https://github.com/dalek-cryptography/bulletproofs.

[Dalb] *curve25519-dalek: A pure-Rust implementation of group operations on Ristretto and Curve25519*. URL: https://github.com/dalek-cryptography/curve25519-dalek.

[Dalc] *x25519-dalek: A pure-Rust implementation of x25519 elliptic curve Diffie-Hellman key exchange*. URL: https://github.com/dalek-cryptography/x25519-dalek.

[Ecc] *Electric Coin Company*. URL: https://electriccoin.co/.

[Ele] *Confidential Transactions*. URL: https://elementsproject.org/features/confidential-transactions.

[Ens] *Ethereum Name Service*. URL: https://ens.domains/.

[FS86] Amos Fiat and Adi Shamir. "How To prove yourself: Practical solutions to identification and signature problems". In: *CRYPTO '86*.

[FST10] David Freeman, Michael Scott, and Edlyn Teske. "A taxonomy of pairing-friendly elliptic curves". In: *JoC '10*.

[GKLRSW21] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. "Reinforced concrete: A fast hash function for verifiable computation". In: *IACR ePrint '21*.

[GKR08] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. "Delegating computation: Interactive proofs for muggles". In: *STOC '08*.

[GKRRS21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems". In: *USENIX Security '21*.

[GLSTW21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. "Brakedown: Linear-time and post-quantum SNARKs for R1CS". In: *IACR ePrint '21*.

[GOPTT22] Chaya Ganesh, Claudio Orlandi, Mahak Pancholi, Akira Takahashi, and Daniel Tschudi. "Fiat–Shamir Bulletproofs are non-malleable (in the algebraic group model)". In: *EUROCRYPT '22*.

[GT03] Shafi Goldwasser and Yael Tauman. "On the (in)security of the Fiat-Shamir paradigm". In: *FOCS '03*.

[GWC19] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. "PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge". In: *IACR ePrint '19*.

[Gro16] Jens Groth. "On the size of pairing-based non-interactive arguments". In: *EUROCRYPT '16*.

[Hal] *Halo2*. URL: `https://github.com/zcash/halo2`.

[Jan] Ján Jančár. *Tool for generating elliptic curve domain parameters*. `https://github.com/J08nY/ecgen`.

[KPS18] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. "xJsnark: A framework for efficient verifiable computation". In: *S&P '18*.

[Lip03] Helger Lipmaa. "On Diophantine complexity and statistical zero-knowledge arguments". In: *ASIACRYPT '03*.

[Loo] *Circuit documentation of Loopring v3 circuit*. URL: `https://github.com/Loopring/protocols/blob/release_loopring_3.6.3/packages/loopring_v3/circuit/statements.md`.

[MBKM19] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. "Sonic: Zero-knowledge SNARKs from linear-size universal and updateable structured reference strings". In: *CCS '19*.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. "Zerocoin: Anonymous distributed E-Cash from Bitcoin". In: *S&P '13*.

[MP03] Daniele Micciancio and Erez Petrank. "Simulatable commitments and efficient concurrent zero-knowledge". In: *EUROCRYPT '03*.

[MSZ21] Simon Masson, Antonio Sanso, and Zhenfei Zhang. "Bandersnatch: a fast elliptic curve built over the BLS12-381 scalar field". In: *IACR ePrint '21*.

[Mer] *Merlin: Composable proof transcripts for public-coin arguments of knowledge*. URL: https://docs.rs/merlin/latest/merlin/.

[Noa] *Findora's cryptographic library*. URL: https://github.com/FindoraNetwork/noah.

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. "Scaling verifiable computation using efficient set accumulators". In: *USENIX Security '20*.

[PBFMW18] Andrew Poelstra, Adam Back, Mark Friedenbach, Gregory Maxwell, and Pieter Wuille. "Confidential assets". In: *FC '18*.

[Par] *Parameter Generation*. URL: https://z.cash/technology/paramgen/.

[Pet18] Paige Peterson. *Reducing shielded proving time in Sapling*. URL: https://electriccoin.co/blog/reducing-shielded-proving-time-in-sapling/.

[Poe18] Andrew Poelstra. *[curves] Curve with group order* $2^255 - 19$. URL: https://moderncrypto.org/mail-archive/curves/2018/000992.html.

[Pol] *hermez-account.js*. URL: https://github.com/hermeznetwork/commonjs/blob/master/src/hermez-account.js.

[Rfc] *Deterministic usage of the digital signature algorithm (DSA) and elliptic curve digital signature algorithm (ECDSA)*. URL: https://www.rfc-editor.org/rfc/rfc6979.html.

[SSKK01] Erkay Savaş, Thomas A. Schmidt, and Çetin K. Koç. "Generating elliptic curves of prime order". In: *CHES '01*.

[Sch89] Claus-Peter Schnorr. "Efficient identification and signatures for smart cards". In: *CRYPTO '89*.

[Set20] Srinath Setty. "Spartan: Efficient and general-purpose zkSNARKs without trusted setup". In: *CRYPTO '20*.

[Sol] *Solana: Build crypto apps that scale*. URL: https://solana.com/ecosystem/explore?categories=wallet.

[Sta] *The STARK curve*. URL: https://docs.starkware.co/starkex-v4/crypto/stark-curve.

[WB19] Riad S. Wahby and Dan Boneh. "Fast and simple constant-time hashing to the BLS12-381 elliptic curve". In: *CHES '19*.

[XZS22] Tiancheng Xie, Yupeng Zhang, and Dawn Song. "Orion: Zero knowledge proof with linear prover time". In: *CRYPTO '22*.

[XZZPS19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. "Libra: Succinct zero-knowledge proofs with optimal prover computation". In: *CRYPTO '19*.

[Zcaa] *P&Q Bulletproofs*. URL: https://github.com/ZcashFoundation/GrantProposals-2018Q2/issues/16.

[Zcab] *What is Jubjub?* URL: https://z.cash/technology/jubjub/.

[Zfn] *Zcash Foundation*. URL: https://zfnd.org/.

[Zks] zksync_types :: tx :: primitives :: packed_public_key. URL: https://github.com/matter-labs/zksync/blob/master/core/lib/types/src/tx/primitives/packed_public_key.rs.

[ark] arkworks contributors. *arkworks zkSNARK ecosystem*. URL: https://arkworks.rs.