Optimizing and Implementing Fischlin's Transform for UC-Secure Zero-Knowledge

Yi-Hsiu Chen and Yehuda Lindell

Coinbase, USA

Abstract. Fischlin's transform (CRYPTO 2005) is an alternative to the Fiat-Shamir transform that enables straight-line extraction when proving knowledge. In this work we focus on the problem of using the Fischlin transform to construct UC-secure zero-knowledge from Sigma protocols, since UC security – that guarantees security under general concurrent composition – requires straight-line (non-rewinding) simulators. We provide a slightly simplified transform that is much easier to understand, and present algorithmic and implementation optimizations that significantly improve the running time. It appears that the main obstacles to the use of Fischlin in practice is its computational cost and implementation complexity (with multiple parameters that need to be chosen). We provide clear guidelines and a simple methodology for choosing parameters, and show that with our optimizations the running-time is far lower than expected. For just one example, on a 2023 MacBook, the cost of proving the knowledge of discrete log with Fischlin is only 0.41ms (on a single core). We also extend the transform so that it can be applied to batch proofs, and show how this can be much more efficient than individually proving each statement. As a contribution of independent interest, we present a new algorithm for polynomial evaluation on any series of sequential points that does not require roots of unity. We hope that this paper will both encourage and help practitioners implement the Fischlin transform where relevant.

1 Introduction

Sigma protocols. A Sigma protocol [Dam10, HL10] is a three-round public-coin honest-verifier zero-knowledge proof with an additional property called special soundness. This property states that given any pair of accepting transcripts (a, e, z) and (a, e', z') with the same first prover message, it is possible to extract the witness. Sigma protocols are extremely ubiquitous, and many natural zero-knowledge proofs can be constructed as Sigma protocols. One of the major advantages of Sigma protocols is that proving that the protocol is also a proof of knowledge is extremely easy, in contrast to the standard definition of a proof of knowledge which is hard to navigate. In addition, it is possible to automatically compile Sigma protocols into full-blown zero-knowledge proofs and zero-knowledge proofs of knowledge with relative ease.

The Fiat-Shamir and Fischlin transforms. The Fiat-Shamir transform [FS86] is the standard way of constructing practical non-interactive zero-knowledge proofs from Sigma protocols in the *random-oracle model*. It works simply by hashing the statement and first prover message, and taking the result as the verifier challenge. In the random-oracle model, this makes sense as taking the verifier query from the random-oracle query is no different from receiving it from the verifier (with the exception that the prover can try

E-mail: yihsiuc@pm.me (Yi-Hsiu Chen), yehuda.lindell@gmail.com (Yehuda Lindell)

Date of this document: 2024-04-04.

many times until it succeeds). The proof obtained using this transform is zero knowledge (in the programmable random-oracle model) and is a proof of knowledge, as long as the extractor is able to *rewind* the adversarial prover. As a result, proofs obtained in this way are not concurrently composable, and in particular are not UC secure [Can01].

In order to overcome this, Fischlin [Fis05] presented a different transform, also in the random-oracle model, that does not require the extractor to rewind the adversary. The ingenious idea behind this transform is to force the prover to attempt many hashes in order to obtain a valid proof, but requiring the output of the hash to have a certain format. This essentially means that the prover "rewinds itself", since the extraction strategy is to rewind the adversarial prover until the extractor obtains two responses to a given query. By the special soundness property, this suffices to obtain the witness. Fischlin's strategy is the same, except that the prover itself – during real proof generation – needs to already query the hash (random oracle) many times in order to generate a valid proof. Thus, the extractor can obtain the witness by just "eavesdropping" on the prover's queries.

Fischlin in practice. In practice, it appears that people shy away from using the Fischlin transform. One may conjecture that this is because it is perceived as being both complicated and expensive. Regarding computational cost, in a horrifically unscientific Twitter poll, but one that certainly provides valuable anecdotal evidence, we asked cryptographers if they use Fiat-Shamir or Fischlin: 163 voted, with **84%** saying that they use Fiat-Shamir. We then asked why Fiat-Shamir if that's the case and **62.8%** said that it's because Fischlin is too expensive (43 voted). We also asked cryptographers to guess how long it takes to run Fischlin for discrete log over secp256k1 on a regular laptop, and only 15.9% said less than 1ms, with 36.4% saying 2-5ms and 27.3% saying greater than 5ms (44 voted), whereas in reality, as we show, it is much less.

Regarding it being complicated, for just one example, the transform allows a lot of freedom in choosing the different parameters, but this becomes very confusing. What parameters should be chosen in practice? Does it change depending on the computing environment? This already makes it hard to implement.

Our contributions. In this paper, we study the Fischlin transform with an eye on *implementation* and *use in practice*. We have the following contributions:

1. We show that one of the parameters (S) in the transform can be completely removed (to be more exact, fixed to always equal 0), at no cost in practice
2. We present a very simple description of the transform that is easy to use as a reference for implementations
3. We describe some very important optimizations for proving and verifying, that significantly impact performance
4. We show how to analytically find optimal parameters for whatever computing platform is being used
5. We provide an extension of Fischlin's transform to n-special soundness (where n accepting transcripts are needed to extract rather than 2) and show that it can be used to generate batch proofs (e.g., knowing the discrete log of many elements) far more efficiently than separately proving each statement

We present experimental results for running times on three different computing environments: a MacBook Pro 2019 with an Intel 2.3 GHz i9 CPU, Web Assembly (WASM, with Node.js V21) on the same MacBook, and a 2023 MacBook Pro with an Apple M3 Pro CPU. Our experiments show that on the aforementioned M3, proving knowledge of a discrete log over secp256k1 takes only **0.41ms** and verifying a proof takes only **0.45ms**, with 128-bit security. This shows that the performance penalty is insignificant in many cases, and certainly in most MPC protocols where zero-knowledge proofs of knowledge are needed. In addition, the time for plain Fiat-Shamir on the same M3 is 0.027ms, and so Fischlin is 15 times slower than plain Fiat-Shamir. Although this is a significant multiple, it is far less overhead than what people seem to believe.

A new algorithm for polynomial evaluation of sequential points. Of independent interest, we present a new algorithm that can be used to compute sequential points on a degree-d polynomial at an amortized cost of *just d additions per point.* To be more exact, given d sequential points $p(a), p(a+1), \ldots, p(a+d)$ on a degree-d polynomial $p(x)$, the algorithm carries out d^2 initial additions and from them on can compute each additional point $p(a+d+1), p(a+d+2), \ldots$ at the cost of just d additions each. We believe that this algorithm may be of independent interest. An application that comes to mind immediately is computing Shamir secret shares for a t-of-n secret sharing, where n is very large.

Motivation. Our motivation for carry out this research was simple: in all of our MPC implementations at Coinbase, we require UC security, and thus simulators must be non-rewinding. When we need a zero-knowledge proof that does not need to be a proof of knowledge, then we use the Fiat-Shamir transform applied to a Sigma protocol. This suffices since the zero-knowledge property does not require rewinding, and likewise soundness (without extraction) is also straight-line. However, when we need a proof of knowledge, we cannot use Fiat-Shamir since it requires rewinding. In such cases, we use the Fischlin transform, and so achieving high efficiency with a simple implementation is paramount.

Related work. In [DV14], the authors study Fischlin's transform in a different context. First, they show that when considering it as a signature scheme, the bounds obtained are far better than for Fiat-Shamir [FS86] since the forking lemma [PS96] is not needed. As such, the efficiency of signatures generated this way can actually be more efficient than by Fiat-Shamir, for the same level of security as provided by the security bounds. Second, they show that better leakage-resilient signatures can be generated using the Fischlin transform. Another work of relevance is [KS22] who focus on improving Fischlin's transform in orthogonal ways to this paper. First, they consider the problem of aggregate signatures; one step in their construction is related to our batch work in the sense that they also apply Fischlin to a batch discrete log proof, and they consider a polynomial evaluation algorithm that is very efficient but requires an n'th root of unity. They also prove a tighter bound on the number of hash computations required and extend the applicability of Fischlin's transform to also include cases where unique prover responses are not required (like in the OR proofs of [CDS94]). Finally, [CGKN21] presents an extension of Fischlin's transform specifically for aggregate signatures that is reminiscent of our generalization for batch proofs. The main difference is that our focus is on zero-knowledge proofs in general and not signature aggregation.[1]

Organization. In Section 2 we present preliminaries, our slightly simplified Fischlin transform, and important implementation details that include optimizations that are very significant for performance. Next, in Section 3 we present the bounds for completeness and soundness, as well as the expected cost, showing that our slight simplification yields much simpler bounds that are easier to understand (we believe that this is important since it removes one of the obstacles for potential implementers). In Section 4 we show how to choose parameters and present experimental results. In Section 5 we describe our

[1] Although our running example throughout the paper is for the proof of knowledge of discrete log which is very related to signatures and aggregation, our focus is the context of zero-knowledge, and we consider other examples in Appendix C.

generalization to n-special soundness and show how it can be applied to batch proofs. We present algorithms for speeding up the computation of these proofs, and show that they are significantly more efficient than running in parallel. Throughout, we use the simple Schnorr proof of discrete log as a running example, but in Appendix C we consider two more complex examples and show that the performance for these is also excellent.

2 The Fischlin Transform (Slightly Simplified)

2.1 Preliminaries

We first rewrite the transform with our notation and an early break optimization. Note that we replace the notation r in [Fis05] with ρ. We denote the computational security parameter by κ_c (in practice, we take 128 by default for 256-bit curves, since this maintains 128-bit security), and we denote the statistical security parameter by κ_s (in practice, we use $\kappa_s = 64$), enabling a 2^{-64} failure probability. We sometime use the notation $[n]$ to denote the set $\{1, \ldots, n\}$.

Sigma protocol: As described above, a Sigma protocol is a three-round public-coin honest-verifier zero-knowledge proof system with special soundness. We formally define the syntax of a Sigma protocol, as follows. Upon common input x and witness w:

1. Let $(m, \sigma) \leftarrow \mathsf{ProverFirstMessage}(x, w)$ denote the first prover message, where m is the message sent by the prover and σ is its local state used to compute the second prover message
2. Let $e \in \{0,1\}^\kappa$ denote the public-coin verifier challenge, of some length κ (in the Fischlin transform, we will use challenges of length t, determined below)
3. Let $z \leftarrow \mathsf{ProverSecondMessage}(x, w, \sigma, e)$ denote the prover response
4. Let $\mathsf{VerifyProof}(x, m, e, z) \in \{0,1\}$ be the final $\mathsf{accept/reject}$ output of the verifier

Throughout, we denote the transcript of the Sigma protocol by (m, e, z). We say that (m, e, z) is an $\mathsf{accepting\ transcript}$ for x if $\mathsf{VerifyProof}(x, m, e, z) = 1$. Since it is crucial to this paper, we define the notion of special soundness.

Definition 1 (special soundness)**.** A Sigma protocol has $\mathsf{special\ soundness}$ for a relation $\mathcal{R}$ if there exists a polynomial-time extractor K that given any x and any pair of transcripts (m, e, z) and (m, e', z') such that $\mathsf{VerifyProof}(x, m, e, z) = \mathsf{VerifyProof}(x, m, e', z') = 1$ and $e \neq e'$, it holds that K outputs w such that $(x, w) \in \mathcal{R}$.

The Fischlin transform. We refer the reader to [Fis05] for full details; we briefly review the idea here. The Fiat-Shamir transform works by having the prover compute the verifier query $e = \mathcal{H}(x, m)$ where x is the statement and m is the first prover message, and where $\mathcal{H}$ is a random oracle. In order to extract the witness from a prover, the knowledge extractor for Fiat-Shamir works by invoking the prover and obtaining an accepting transcript (m, e, z). Then, the extractor rewinds the prover and replaces the random-oracle reply e on input (x, e) with some $e' \neq e$ and hopes to receive a different accepting transcript (x, m, e', z') with the same m. If yes, then it succeeds in extracting the witness, by the special soundness property. Thus, the idea is to obtain two different random oracle responses queries to the same input (x, m). Fischlin achieves this differently by having the prover search for an e such that the hash of the *entire transcript* has a specified format of b leading zeroes (for some parameter b). That is, the prover generates the first message m and then looks for e and z so that (m, e, z) is an accepting transcript *and* $\mathcal{H}_b(x, m, e, z) = 0$, where $\mathcal{H}_b$ denotes the output of $\mathcal{H}$ truncated to b bits. Since the prover is expected to need 2^b queries in order to succeed for this, if it makes 2 queries to $\mathcal{H}_b$ with valid transcripts for the same (x, m) and different values of e, then the extractor will be able to extract the witness, again by the special soundness. However, this strategy doesn't work since the honest-verifier zero-knowledge property of Sigma protocols states that given e ahead of time, it is possible to find (m, z) such that (m, e, z) is an accepting transcript (without knowing the witness). As such, a cheating prover can guess e, find (m, z) such that (m, e, z) is an accepting transcript and hope that $\mathcal{H}_b(x, m, e, z) = 0$. It will succeed with probability 2^{-b} in every trial and so will be able to cheat in time 2^b.

In order to prevent this attack, we cannot increase b so that 2^b is super-polynomial, since then an honest prover would not be able to prove the proof in polynomial time. Instead, Fischlin's transform works by simultaneously running this ρ times in parallel. That is, the prover first generates ρ first messages $\vec{m} = (m_1, \ldots, m_\rho)$ and then works one proof at a time to find e_i, z_i such that (m_i, e_i, z_i) is an accepting transcript *and* $\mathcal{H}_b(x, m_i, e_i, z_i) = 0$. This prevents the prover from guessing e and then going back to find (m, z) since the entire vector $\vec{m}$ of ρ first messages is included. Furthermore, since in most Sigma protocols (and this is needed in Fischlin's transform as pointed out by [KS22]), the third message z is fully determined by m and e, the prover must be lucky and guess all $e_1, \ldots, e_\rho$ ahead of time. However, the chances of this happening is now $\left(2^{-b}\right)^\rho$ and so the prover would need to run expected time $2^{b \cdot \rho}$ in order to cheat. By setting $\rho \cdot b \geq \kappa_c$, it follows that this is infeasible.

2.2 A Slightly Simplified Fischlin Transform

The original transform by Fischlin [Fis05] did not actually require that all random-oracle outputs equal zero. Rather, some slack was allowed in the form of a bound S. This slack improves completeness, but as we show below, this is not required. We therefore present a simplified version of the Fischlin transform below. Note that our presentation is in the form of a clear specification in order to make implementation easy. We do leave the choice of parameters open here; below we show how these should be chosen.

- **Parameters:**
  - ρ: the total number of "parallel" repetitions conducted
  - b: the output size in bits of the random oracle (this is the difficulty to find a valid challenge in each repetition)
  - t: for the prover only, and is the maximum number of challenges in each repetition
- **Input:**
  - *Common input:* x
  - *Private prover input:* w such that $(x, w) \in R$, where R is the relation being proven.
- **Prover:** $\pi \leftarrow \mathsf{proveFischlin}_{\rho,b,t}(\mathsf{ProverFirstMessage}, \mathsf{ProverSecondMessage}, x, w, \mathsf{sid})$
  1. For $i = 1, \ldots, \rho$,
     (a) compute $(m_i, \sigma_i) \leftarrow \mathsf{ProverFirstMessage}(x, w)$ independently for each i
  2. Let $\vec{m} = (m_1, \ldots, m_\rho)$
  3. $\mathsf{common\text{-}h} \leftarrow \mathcal{H}(x, \vec{m}, \mathsf{sid})$
     (This is a full hash, with output length $2 \cdot \kappa_c$.)
  4. For $i = 1, \ldots, \rho$:

```
    (a) For $e_i = 0, \ldots, 2^t - 1$
        i. $z_i \leftarrow \mathsf{ProverSecondMessage}(x, w, \sigma_i, e_i)$
        ii. $h_i \leftarrow \mathcal{H}_b(\mathsf{common\text{-}h}, i, e_i, z_i)$, where $\mathcal{H}_b$ is the first $b$ bits of output of $\mathcal{H}$
        iii. If $h_i = 0$, break
        iv. If $e_i = 2^t - 1$, redo the entire proof from the beginning
            (If this occurs, then it means that no break ever took place, meaning that the proof failed.)
5. $\vec{e} \leftarrow (e_1, \ldots, e_\rho)$
6. $\vec{z} \leftarrow (z_1, \ldots, z_\rho)$
7. $\pi \leftarrow (\vec{m}, \vec{e}, \vec{z})$
8. Output $\pi$
```

- **Verifier:** $(\mathsf{accept}/\mathsf{reject}) \leftarrow \mathsf{verifyFischlin}\rho, b(\mathsf{VerifyProof}, x, \pi, \mathsf{sid})$

```
1. Parse $\pi$ as $(\vec{m}, \vec{e}, \vec{z})$
2. If $\vec{m}$, $\vec{e}$, and $\vec{z}$ do not each have $\rho$ elements, then output reject
3. $\mathsf{common\text{-}h} \leftarrow \mathcal{H}(x, \vec{m}, \mathsf{sid})$
4. For $i \in \{1, \ldots, \rho\}$
    (a) Halt and output reject if $\mathsf{VerifyProof}(x, m_i, e_i, z_i) = 0$
    (b) Halt and output reject if $\mathcal{H}_b(\mathsf{common\text{-}h}, i, e_i, z_i) \neq 0$
5. Output accept
```

We note that the separation of the computation of $\mathcal{H}$ into common-h and the rest is an optimization discussed in [Fis05] to reduce the cost of hashing. Observe that in a Merkle-Damgård hash, this is identical to preprocessing some blocks in a hash and computing the rest at a later time, which clearly has no impact on security. However, even for other hash functions, and in particular for random oracles, it is easy to show that this has the same effect (as long as the output of $\mathcal{H}$ is long enough).

Security. The above transform was proven secure in [Fis05]; the fact that we take $S = 0$ makes no difference since it is a valid parameter in the original transform. Below, we do show that one can derive much simpler completeness and soundness bounds for this case.

2.3 Implementation Issues and Optimizations

Implementing flexible parameters. One of the challenges when implementing the Fischlin transform is that there is no single good choice of parameters. Below, we will show how to choose parameters in a clear way. However, in practice, the best choice of parameters may change over time and as we will see can depend on the computing platform being used. However, changing parameters breaks backward compatibility in the sense that changing the parameters at the prover means that an older verifier fails, and vice versa. This makes updating parameters painful for software that is in use. In addition, this makes it impossible for provers on different platforms, for whom different parameters are optimal, to work with a single verifier. In order to overcome this difficulty, we have the prover include the parameters b and ρ as part of the proof itself. In order to ensure that soundness holds, this requires that the verifier first check that $\rho \cdot b \geq \kappa_c$, and only then proceed. Note that t need not be sent, since soundness does not depend on t, and it is not used by the verifier anywhere.

Regarding security, we need to show that the ability to choose b and ρ flexibly does not make it easier to cheat. First, we remark that for every ρ there exists a unique smallest

$b_{\min}$ such that $\rho \cdot b_{\min} \geq \kappa_c$. Furthermore, any proof that is valid for ρ and $b > b_{\min}$ is also valid for ρ and $b_{\min}$, and any proof that is valid for $\rho > \kappa_c$ is also valid for $\rho = \kappa_c$ since $b \geq 1$ is always.[2] This implies that there are at most κ_c possible choices of (b, ρ). Thus, an adversary that breaks soundness with probability ε when it can choose any (b, ρ) that it likes can easily be converted into an adversary that break soundness for a specific choice of (b, ρ) for which $\rho \cdot b \geq \kappa_c$ with probability $\frac{\varepsilon}{\kappa_c}$. This is a very small reduction in security (at most $\log \kappa_c$ bits). We stress that it *is* crucial that the verifier check that $\rho \cdot b \geq \kappa_c$, or the proof is trivially broken (by the prover setting $\rho = b = 1$).

Optimizing proof generation. Since the second prover message is computed many times, computing it from scratch each time can add significant overhead. For example, consider the case of the proof of knowledge for discrete log (the classic Schnorr proof), the Sigma protocol is as follows:

1. *Statement:* a group element Q with witness w (i.e., $Q = w \cdot G$)
2. $\mathsf{ProverFirstMessage}$: choose a random $r \in_R \mathbb{Z}_q$ and compute $R = r \cdot G$
3. $\mathsf{ProverSecondMessage}$: compute $z = r + e \cdot w \bmod q$
4. $\mathsf{VerifyProof}$: verify that Q, R are group elements, and that $z \cdot G = R + e \cdot Q$

A naive implementation of the prover would run $\mathsf{ProverSecondMessage}$ from scratch for every iteration, requiring an expected $\rho \cdot 2^b$ modular multiplications and additions. However, observe that the internal loop is over a specific set of values e_i, and in particular over $e_i = 0, \ldots, 2^t - 1$. As a result, given $z_i = r + e_i \cdot w \bmod q$, it is possible to compute

$$z_{i+1} = r + e_{i+1} \cdot w = r + (e_i + 1) \cdot w = (r + e_i \cdot w) + w = z_i + w \bmod q.$$

Thus, in each iteration, it suffices to carry out a single modular addition rather than a modular multiplication and addition. This is very significant. Furthermore, due to the linear structure of many (most) Sigma protocols, this type of optimization is typically possible.

Optimizing proof verification. The proof verification involves verifying ρ copies of the basic Sigma protocol. Rather than verifying each proof independently, we can use batch verification techniques, and in particular the small exponents test in [BGR98]. This method generates a random linear combination of the different equations, and verifies equality of the result (details below). In addition, Sigma protocol verification often includes validity checks on the statement (e.g., it is a valid subgroup element), and such checks should only be carried out once. We demonstrate the batch verification optimization for the proof of knowledge of discrete log proof, described above, for *prime-order groups* (which are anyway what are used in practice). For this proof, for every i, the verifier needs to check that $z_i \cdot G = R_i + e_i \cdot Q$. These equalities can be batch-verified more efficiently by constructing random linear combinations, and verifying equality only of the sum. In order to illustrate this technique on a simpler example, consider the task of verifying that $A_i = a_i \cdot B$ for $i = 1, \ldots, \rho$. This would cost ρ full-length elliptic-curve multiplications. However, if instead we chose random values σ_i of length s and computed $\sigma_i \cdot A_i$ and finally verified $\sum_{i=1}^{\rho} \sigma_i \cdot A_i = [\sum_{i=1}^{\rho} \sigma_i \cdot a_i \bmod q] \cdot B$, then the cost would be ρ *short* elliptic-curve multiplications and one full-length multiplication. If the length s of σ_i is a quarter of the length of the order q of the curve, then this is about $1/4$ of the overall cost. Intuitively, security holds since if one of the equalities doesn't hold, then the probability that this inequality will be cancelled out with a random choice of σ_i is 2^{-s}.

In more detail, for the specific case of discrete log proof verification ($z_i \cdot G = R_i + e_i \cdot Q$), the batch verification is carried out as follows:

[2] A proof with a larger ρ won't be accepted for a smaller ρ since the lengths won't match. However, we can modify the verifier so that it ignores anything beyond κ_c, and the same soundness bounds hold.

1. Choose random $\sigma_1, \ldots, \sigma_\rho \in_R \{0, \ldots, 2^{\kappa_s} - 1\}$
2. Compute $z_{\text{sum}} = \sum_{i=1}^{\rho} \sigma_i \cdot z_i \bmod q$, $e_{\text{sum}} = \sum_{i=1}^{\rho} \sigma_i \cdot e_i \bmod q$ and $R_{\text{sum}} = \sum_{i=1}^{\rho} \sigma_i \cdot R_i$
3. Verify that $R_{\text{sum}} = z_{\text{sum}} \cdot G - e_{\text{sum}} \cdot Q$

The proof of this method (why it's correct and secure) is standard, and appears in Appendix A. Regarding *efficiency*, the cost of this optimized verification is 2 full elliptic-curve multiplications (EC-MULT) and ρ short EC-MULTs of length κ_s each. In the case of 256-bit curves and $\kappa_s = 64$, this means that a short EC-MULT is 1/4 of the cost of a full EC-MULT. With these parameters, the overall cost is reduced from ρ EC-MULTs (and ρ very short EC-MULTs) to $\rho/4 + 2$ EC-MULTs, which is one quarter of the time.

An interesting point to note here is that standard EC-MULT in good libraries (like OpenSSL) is *constant time*. This means that computing $\sigma_i \cdot R_i$ for a 64-bit value σ_i will be the same cost as for a 256-bit value. Thus, in order to full utilize this optimization, a non-constant time EC-MULT must be implemented and used. Note that this is not an issue at all for zero-knowledge verification, since all values are public. We also remark that without such a non-constant time EC-MULT, the plain verification of $R_i = z_i \cdot G - e_i \cdot Q$ will actually cost two *full* EC-MULTs, even though e_i is extremely short (e.g., 9 bits). Thus, a non-constant time operation has a significant impact here. For example, for secp256k1, the time to verify with the above optimization using constant-time EC-MULT is 1.2ms, but the time to verify using non constant-time EC-MULT is 0.58ms.

Experimental results. We compare the best running times for a naive implementation (with the best parameters for that implementation), where by naive we mean without the optimizations demonstrated in Section 2.3. The parameters that we use for the naive proof are $b = 3$ and $\rho = 43$ for both secp256k1 and Ed25519, and for the optimized proof are: $b = 4$ and $\rho = 32$ for secp256k1, and $b = 3$ and $\rho = 43$ for Ed25519. In Sections 3 and 4 we discuss these choices of parameters in depth. We show the results of our experiments on a 2019 MacBook Pro with a 2.3 GHz 8-Core Intel i9 CPU and on a 2023 MacBook Pro with an Apple M3 Pro CPU.

Table 1: Running times for Fischlin for the discrete log relation, comparing our optimized implementation to a straightforward one. The times are in milliseconds, averaging over 1,000 executions. Intel and M3 are as described above.

	Prover			**Verifier**		
	Naive	**Optimized**	**Gain**	**Naive**	**Optimized**	**Gain**
Intel – secp256k1	1.13	0.93	18%	0.98	0.58	41%
Intel – Ed25519	0.96	0.76	21%	1.86	1.62	13%
M3 – secp256k1	0.51	0.42	18%	0.47	0.36	23%
M3 – Ed25519	0.57	0.45	21%	1.35	1.12	17%

We stress that the comparison for the verifier time for the case of secp256k1 shows the running times for different parameters – for naive we use $\rho = 43$ whereas for optimized we use $\rho = 32$. This is significant since the running time for verification depends primarily on ρ. Thus, although the verification is indeed 41% faster, it is important to note that this is also due to the ability to use different parameters and not just because of the optimized verification procedure. If we compare the verification time for $\rho = 32$ for both the naive and optimized verification on the Intel machine, we obtain times of 0.75 and 0.58, respectively, demonstrating a 22% improvement that *is* due to the batch verification method. Note that the verification times for Ed25519 are significantly slower than expected since our current implementation does not support *non-constant time* multiplication, and so the cost of computing $e_i \cdot Q$ is the same as if e_i was a full-size element of $\mathbb{Z}_q$.

Prover versus verifier cost. It is worth noting that as b gets bigger and ρ gets smaller, the cost of verification gets lower. This is because the verification does not require multiple hashing and is just a function of ρ. As a result, it may sometimes make more sense to use parameters that give a higher prover time but a lower overall prove-and-verify time. This depends on the protocol in question (e.g., each party generates one proof and verifies one other, or maybe each party generates one proof and verifies 5 other). This is demonstrated below for our optimized implementation.

Table 2: Running times for the discrete log proof with different parameters (ρ is taken as $\lceil 128/b \rceil$ always), on a 2019 MacBook Pro with a 2.3 GHz 8-Core Intel i9 CPU. Times are in milliseconds, averaged over 1,000 executions.

	Prover				**Verifier**			
b	3	4	5	6	3	4	5	6
secp256k1	0.94	0.93	1.07	1.47	0.75	0.58	0.48	0.41
Ed25519	0.76	0.77	1.00	1.37	1.62	1.23	1.03	0.9

To illustrate what the optimal choice would be for secp256k1, in a protocol where each party generates one proof and verifies one proof the best choice of parameters would be $b = 4$ (since that minimizes the *sum* of the prover and verifier time), whereas in a protocol where each party generates one proof and verifies 5 proofs the best choice of parameters would be $b = 5$. As above, the Ed25519 verification times are slower than they should be due to our library currently only supporting constant-time operations.

3 Theoretical Security Bounds and Cost

The proof of security of Fischlin's transform appears in [Fis05]. In this section, we review those bounds, and show that our simplification of the transform (that fixes $S = 0$) provides far simpler bounds that are also more simple to understand.

Let s_m and s_z denote the size of m and z in the Sigma protocol. For the security bounds and complexities of Fischlin transform, we let $\varepsilon_{\text{comp}}$ denote the completeness error and let $\varepsilon_{\text{sound}}$ denote the (special) soundness error.

We provide a novel analysis of the Fischlin transform below for the case that $S = 0$ (meaning that the prover needs to find hashes that equal 0 for all iterations). We show that this significantly simplifies the analysis, without significantly impacting efficiency. We then show that these bounds are the same as proven by [Fis05] for this special case, and thus the proof remains the same.

Completeness error. In the transform we describe above, formally the completeness error is 0 since the prover just repeats until it succeeds (see Step 4(a)iv). Thus, what we really mean by completeness error here is the probability that the prover fails in a single attempt and thus needs to repeat the proof. In our simplified transform, the failure probability in each iteration (of $i = 1, \ldots, \rho$) is at most $e^{-2^{t-b}}$. This is because the prover fails unless it finds $h_i = 0$. Now, for any given e_i the probability that h_i equals 0 is 2^{-b} and there are 2^t attempts. This implies that the prover fails with probability

$$\left(1 - 2^{-b}\right)^{2^t} = \left[\left(1 - 2^{-b}\right)^{2^b}\right]^{2^{t-b}} < e^{-2^{t-b}}.$$

The prover needs to succeed in all iterations, and therefore the probability that there is any failed iteration can be bounded using the union bound over all ρ iterations. This yields the following completeness error bound

$$\varepsilon_{\text{comp}} \leq \rho \cdot e^{-2^{t-b}} = 2^{-2^{t-b} \log e + \log \rho}.$$

For the concrete parameters that we use (fixing $t = b + 5$ and $\rho \leq 64$; discussed more below), the completeness error is at most 2^{-40}, meaning that a proof would need to be repeated with an extremely low probability. Furthermore, even if such a repeat is needed (an expected once every 2^{40} proofs), this has very little impact on efficiency since it merely requires a second attempt (the probability that a third attempt is needed is 2^{-80}).

The above argument can be easily formalized, but we show now that it is actually identical to that of [Fis05] for the case that $S = 0$. Fischlin [Fis05, Page 10] showed that the completeness bound is $\rho \cdot e^{-(S+1)2^{t-b}} + S \cdot (\rho - 1) \cdot e^{\rho \ln(e(2S+1))} \cdot e^{-(S+1)2^{t-b}}$.[3] Plugging in $S = 0$, we have that this equals $\rho \cdot e^{-2^{t-b}}$, which is exactly our bound above.

Soundness error. Regarding soundness, the only way that the prover can succeed in proving without querying $\mathcal{H}$ with the same $(x, \vec{m}, i)$ twice is to obtain $h_i = 0$ for every $i \in [\rho]$ with a single hash computation. The probability of this occurring is 2^{-b} for each $i \in [\rho]$ and so the probability that this happens for every i is $\left(2^{-b}\right)^{\rho}$, meaning that the soundness error for a fixed prefix $(x, \vec{m})$ is $\varepsilon_{\text{sound}} \leq 2^{-\rho \cdot b}$. As shown in [Fis05], we can therefore conclude that for an adversary making Q queries to the random oracle with any prefix $(x, \vec{m}, \ldots)$, the soundness is bounded by

$$\varepsilon_{\text{sound}} \leq (Q + 1) \cdot 2^{-\rho \cdot b}.$$

As for completeness, we show that this is exactly the same bound as in [Fis05] for the case of $S = 0$. Specifically, [Fis05, Page 12] shows that $\varepsilon_{\text{sound}} \leq (Q+1) \cdot (S+1) \cdot \binom{S+\rho-1}{\rho-1} \cdot 2^{-\rho \cdot b}$. Plugging in $S = 0$, we have that this equals $(Q+1) \cdot 2^{-\rho \cdot b}$ which is exactly the same bound that we obtained.

Proof size. The size of the proof, $\pi = (\vec{m}, \vec{e}, \vec{z})$, is $\rho \cdot (s_m + t + s_z)$, where s_m and s_z denote the size of the first and third prover messages m and z, respectively. (When using the flexible parameter method, the proof also includes the values b, ρ, but these are small.) It is important to note that b does not impact the size of the proof, and it depends only on ρ. This means that optimizing for proof size would potentially provide a different trade-off than for proof time.

Expected number of hashes computed by the prover. Considering the case of $S = 0$, the prover succeeds in an iteration if it finds a hash that equals 0. Given that the hash output is of length b bits, it follows that each hash equals 0 with probability 2^{-b} and so the expected number of hash computations in a single iteration is 2^b and the expected number of hash computations overall is $\rho \cdot 2^b$. We note that the prover may need to compute the proof from scratch if it fails, but this happens with probability $\varepsilon_{\text{comp}}$, and the expected number of iterations is $\frac{1}{1-\varepsilon_{\text{comp}}}$. Thus, the overall expected number of hash computation is $\frac{1}{1-\varepsilon_{\text{comp}}} \cdot \rho \cdot 2^b$, which is very close to $\rho \cdot 2^b$ for the parameters that we use.

Summary of the bounds. We summarize the bounds in the following table:

[3] The notation we use is the same, except that we use ρ for the value r in [Fis05].

Table 3: Summary of the original bounds in [Fis05] and the bounds where $S = 0$. For completeness and soundness, for a value s given the bound is 2^{-s}.

Bounds	$S \geq 0$ **(Fischlin)**	$S = 0$
Completeness	$\left[(S+1) \cdot 2^{t-b} \log e\right] - [\rho \log(e(2S+1)) + \log(\rho S)]$	$2^{t-b} \log e - \log \rho$
Soundness	$\rho \cdot \left(b - \log\left(e \cdot \frac{S+\rho}{\rho-1}\right)\right) - \log(S+1)$	$\rho \cdot b$
Expected #hash	$\sim 2^b \cdot \rho$	$\frac{1}{1-\varepsilon_{\text{comp}}} \cdot 2^b \cdot \rho$
Proof size	$\rho \cdot (s_m + t + s_z)$	

On the additional parameter S**.** As our analysis above shows, choosing $S = 0$ versus $S > 0$ yields a *better* soundness bound. In addition, it has no effect on the expected running time of the protocol. Thus, the only reason to choose S to be non-zero is to decrease the completeness error, especially when $t - b$ is small (since the completeness error is $2^{-2^{t-b} \cdot \log e + \log \rho}$). However, in such a case, one can always increase t without impacting the expected running time, which as we showed above does not depend on t (except for the length of e_i which has a minor impact on the efficiency of the hash computation and the prover response).[4] Furthermore, achieving negligible completeness isn't necessary in practice, especially when the only cost is to rerun the protocol an additional time.

4 Optimal Parameters and Experimental Results

As we have discussed, one of the challenges with implementing the Fischlin transform is the need to choose the parameters ρ, b, t (and S in the original transform). One can of course always find the optimal parameters experimentally, but this can be painful since it requires running multiple experiments for every different protocol and every different computing platform (as we will see, the relation between the cost of computing the hash versus the rest of the proof has a major impact on efficiency). In this section, we will provide a straightforward rule for choosing t as a function of ρ and b, and we will show how to analytically estimate the optimal parameters to provide a good choice of parameters with minimal experimentation.

Choosing t **as a function of** ρ **and** b**.** Given a specific choice of ρ and b and a target completeness error $\varepsilon_{\text{comp}} = 2^{-\sigma}$,[5] one can compute t directly so that the completeness error is at most $2^{-\sigma}$. From the bounds we have shown, in order to have completeness $2^{-\sigma}$ we need to set t so that $2^{t-b} \log e - \log \rho \geq \sigma$, which holds if and only if

$$t \geq \log\left(\frac{\sigma + \log \rho}{\log e}\right) + b = \log(\sigma + \log \rho) - \log \log e + b.$$

This can therefore easily be computed. We now make the choice of t even easier by showing that for a concrete completeness error of 2^{-40} we can provide a simple rule depending on ρ alone. Given that $\log \log e \approx 0.529$, we require that $t \geq \log(40 + \log \rho) - 0.529 + b$. Notice that $\log 46 \approx 5.523$ and thus $\log(40 + 6) - 0.529 < 5$. This means that as long as $\log \rho \leq 6$, or equivalently $\rho \leq 64$, it suffices to take $t = b + 5$. Furthermore, it never suffices to take $t = b + 4$ since $\log 40 - 0.529 > 4$. In the other direction, we look at when $t = b + 6$ is not enough. This occurs when $\log(40 + \log \rho) - 0.529 > 6$ which occurs when $\log(40 + \log \rho) > 6.529$ or $\log \rho > 2^{6.529} - 40 > 52$. Thus, $t = 6$ suffices up to $\rho = 2^{52}$, which will never be chosen in practice. We therefore conclude with the following rule:

[4] The length of e_i has a more significant impact on the verification time, since e_i is used to multiply group elements. However, since t is anyway very small, increasing it by a few bits makes no real difference.

[5] Recall that what we mean here is the probability that the proof will fail and needs to be repeated.

Rule for t: for any value of b, for $\rho \leq 64$ set $t = b + 5$ and for $\rho > 64$ set $t = b + 6$.

This rule can be hard-coded and so choosing t is immediate and trivial.

The challenge of choosing ρ and b. The notable trade-off regarding efficiency is due to choosing between b and ρ. Fixing a target soundness error of $2^{-\kappa}$, we have that $\rho \cdot b = \kappa$ (for the case of $S = 0$). This therefore means that the expected running time increases only linearly with ρ and exponentially with b. As such, it may appear that it is better to take b as small as possible and ρ as large as possible. However, this hides the fact that the expected $\rho \cdot 2^b$ iterations are *hash operations* and computing ProverSecondMessage, whereas ProverFirstMessage also needs to be computed ρ times. Note also that in typical Sigma protocols, ProverFirstMessage requires more expensive operations (e.g., for elliptic-curve proofs, it requires multiplying EC points by a scalar – aka group exponentiations), whereas ProverSecondMessage requires only operating on the scalar values themselves. (See the discrete log Sigma protocol example given in Section 2.3 under "optimizing proof generation".) As a result, finding the right trade-off between b and ρ means running experiments to minimize the cost of ρ "expensive" operations versus $\rho \cdot 2^b$ "inexpensive" operations. This trade-off depends on the specific Sigma protocol as well as the computation platform (e.g., it is greatly influenced by whether or not the hash can be computed using hardware acceleration and how expensive the EC operations are which depends on the curve being used and the level of optimizations available).

Analytical parameter optimization. As mentioned above, the cost of generating a proof is dominated by the cost of computing the first prover message ρ times, and then computing $\rho \cdot 2^b$ hash computations and second prover messages. For most Sigma protocols, the cost of computing the second prover message z is small, and we will therefore ignore it in our analysis below. If not, then the equations below need to be updated to include this cost, but the methodology is the same.

Let T_1 denote the cost of computing the first prover message, let T_{hash} denote the cost of computing the hash on the appropriate size, and let a be a constant such that $T_1 = a \cdot T_{\text{hash}}$. We note that the value a can easily be found experimentally on any specific computing device. Then, we have that the cost of computing a proof is

$$T = \rho \cdot T_1 + \rho \cdot 2^b \cdot T_{\text{hash}} = \rho \cdot a \cdot T_{\text{hash}} + \rho \cdot 2^b \cdot T_{\text{hash}} = T_{\text{hash}} \cdot (a \cdot \rho + \rho \cdot 2^b).[6]$$

According to this, given a (found experimentally on the device being used), we have that the cost of proving can be minimized by finding the minimum of the function $T = a \cdot \rho + \rho \cdot 2^b$. To further simplify this to a univariate function, since we know that $b \cdot \rho = \kappa_c$, we can write $T = a \cdot \rho + \rho \cdot 2^{\kappa_c/\rho}$ and find the minimum of this function for $\rho > 0$.

Regarding the cost of verifying, this is naively ρ times the cost of the standard verification of the proof, but this can be optimized using batch verification, as described in Section 2.3. For discrete log, as an example, this cost is dominated by computing $z \cdot G - e \cdot Q$. In this case, e is very short (of length t only) and thus this is dominated by a single exponentiation. Thus, minimizing the verifier cost is simply minimizing ρ. However, in protocols where all parties send and receive proofs, the cost can be minimized by minimizing the *combined cost* of proving X proofs and verifying Y proofs (depending on the protocol) under the constraint that $\rho \cdot b = \kappa_c$. Below, we focus on the prover cost only, with the understanding that it can all be generalized to the overall cost depending on the protocol in a straightforward way.

We will use our running example of a proof of knowledge of discrete log (over two different curves) in order to demonstrate our methodology; other examples are given in

[6] As noted above, if the cost of the second prover message is significant, then the equation becomes $T = \rho \cdot T_1 + \rho \cdot 2^b \cdot (T_{\text{hash}} + T_2)$, where T_2 is the cost of computing the second prover message.

Section C. Recall that the discrete log Sigma protocol is described in Section 2.3. For the knowledge of discrete log proof, the first prover message is a single elliptic-curve multiplication of the group generator by a random scalar. We denote this operation by MUL-G . (Note that MUL-G is often much more efficient than an elliptic curve multiplication of an arbitrary point, since a pre-computed table of $G, 2G, 4G, \ldots$ can be used.) Denoting the cost of MUL-G by T_{mulg}, we have that the cost function is $T = \rho \cdot T_{\text{mulg}} + \rho \cdot 2^b \cdot T_{\text{hash}}$, and so as described above our aim is to find a such that $T_{\text{mulg}} = a \cdot T_{\text{hash}}$ so that we can find the minimum of the function $a \cdot \rho + \rho \cdot 2^b$.

We computed a for two curves – secp256k1 and Ed25519 – on three different computing platforms: a 2019 MacBook Pro with a 2.3 GHz 8-Core Intel i9 CPU, web assembly (WASM) using Node.js V21 on the aforementioned MacBook Pro, and on a 2023 MacBook Pro with an Apple M3 Pro CPU. We note that these calculations and even computing the optimal parameters could theoretically be done at run-time. The results appear in Table 4.

Table 4: Operations costs on different platforms in ms (MUL-G is scalar multiplication of the EC generator point). Intel, WASM and M3 are as described above.

	MUL-G secp256k1	MUL-G Ed25519	SHA256 64 bytes	secp256k1 / SHA256	Ed25519/ SHA256
Intel	0.0146	0.0085	0.000639	23	13
WASM	0.068	0.0051	0.00109	62	47
M3	0.0078	0.008	0.000165	47	48

It is interesting to note that operations behave very differently on different platforms. In our WASM implementation (using Node.js V21), the ratio for Ed25519 to SHA256 is much lower than for secp256k1 to SHA256, but this is reversed in Intel. We would therefore expect in WASM for secp256k1 that the optimum would be fewer exponentiations and more hash computation. Indeed, as we show below, the optimal parameters obtained differ for these platforms.

We used WolframAlpha to find the minimum of each function (for each curve and platform) for $\kappa_c = 128$. In Table 5, we provide the parameters found analytically for the functions. We provide the actual minimums, along with natural number parameters that we derive from them. Note that since we want $\rho \cdot b$ to be as close as possible to $\kappa_c = 128$, we adjust the minimums obtained appropriately.

Table 5: Analytically computed trade-offs for different protocols and different environments for soundness 2^{128}. ρ_{min} denotes the actual function minimum, and ρ denotes the closest approximate choice that is amenable for $\rho \cdot b = 128$.

	secp256k1			Ed25519		
	ρ_{min}	ρ	b	ρ_{min}	ρ	b
Intel	33.6	32	4	38.6	43	3
WASM	26.9	32	4	28.6	32	4
M3	28.6	32	4	28.4	32	4

Experimental results. We now present actual running times for different parameters, demonstrating that our analytic results are very close. Note that since the actual minimum for ρ is not necessarily a valid choice for ρ (since we always want $\rho \cdot b = 128$), our choice as to whether or not to take a larger or smaller ρ is a pure guess. The following results are the average over 1,000 executions, and demonstrate that our analytic approach works very well, and gives a very good estimate of the best parameters to be used.

Table 6: Running times for Fischlin for discrete log; average over 1,000 executions.

	secp256k1				**Ed25519**			
	$\boldsymbol{\rho = 43}$ $\boldsymbol{b = 3}$	$\boldsymbol{\rho = 32}$ $\boldsymbol{b = 4}$	$\boldsymbol{\rho = 26}$ $\boldsymbol{b = 5}$	$\boldsymbol{\rho = 22}$ $\boldsymbol{b = 6}$	$\boldsymbol{\rho = 43}$ $\boldsymbol{b = 3}$	$\boldsymbol{\rho = 32}$ $\boldsymbol{b = 4}$	$\boldsymbol{\rho = 26}$ $\boldsymbol{b = 5}$	$\boldsymbol{\rho = 22}$ $\boldsymbol{b = 6}$
Intel	0.94	**0.93**	1.07	1.47	**0.76**	0.77	1.00	1.37
WASM	3.79	**3.39**	3.54	4.04	3.93	**3.52**	3.53	4.13
M3	0.49	**0.41**	0.42	0.50	0.56	**0.45**	0.45	0.52

5 Fischlin for n-Special Soundness and Batch Proofs

5.1 Extending the Fischlin Transform

The Fischlin transform of [Fis05] applies to Sigma protocols with special soundness, as in Definition 1. In this section, we extend the Fischlin transformation to the more general case where n different accepting transcripts (with the same first message) are needed to extract, rather than 2. As we will show, this is of interest for batch proofs (e.g., proving knowledge of many discrete logs at one). We begin by recalling the definition of n-special soundness.

Definition 2 (n-special soundness)**.** A Sigma protocol has n-special soundness for a relation $\mathcal{R}$ if there exists a polynomial-time extractor K that given any x and any set of n transcripts $T = \{(m, e_i, z_i)\}_{i\in[n]}$ such that all have the same first prover message m, all e_i messages are distinct, and $\mathsf{VerifyProof}(x, m, e_i, z_i) = 1$ for all $i \in [n]$, it holds that K outputs w such that $(x, w) \in \mathcal{R}$.

We now show how to extend the Fischlin transform to achieve straight-line extraction for the case of n-special soundness, rather than regular special soundness. We actually take the exact same transformation (with $S = 0$) but with different parameters. The proofs of completeness and zero-knowledge properties remain unchanged, since they do not utilize the special soundness property. We therefore focus on extraction from here on.

Extraction method. We show how to construct an extractor for the case of n-special soundness. The extractor goes through all the queries and answers in an attempt to obtain n valid random oracle queries that have common $(x, \vec{m}, i)$ and n distinct (e_i, z_i) pairs such that $\mathsf{VerifyProof}(x, m_i, e_i, z_i) = 1$.[7] Once these transcripts are found, we can simply apply the extractor of Definition 2 to extract the witness. Therefore, all we need to do is show that the probability that a successful prover does not query the oracle so that n such accepting transcripts are found is low. Stated differently, we calculate the soundness bounds for this strategy.

If a prover succeeds in proving without querying n accepting transcripts, then for a given $x, \vec{m}$ and for each i, there must be at most $n-1$ distinct challenges $e_1, \cdots, e_{n-1}$ that the prover queried to the random oracle. Given that an individual query returns 0 with probability 2^{-b}, the probability that the prover can find an accepting transcript for a given i with at most $n-1$ hash queries is at most $(n-1)\cdot 2^{-b}$. Since the prover needs to succeed for all $i = 1, \ldots, \rho$ with at most $n-1$ hash queries (since success for any single i suffices

[7] For efficiency purposes, in the actual transform we split the computation of this hash into two parts, but this is inconsequential here.

to extract the witness), the probability that it generates an accepting proof without ever generating n accepting transcripts is at most

$$\left((n-1)\cdot 2^{-b}\right)^{\rho} = 2^{-\rho\cdot(b-\log(n-1))}.$$

As for the case of special soundness as shown by [Fis05], the soundness error for an adversary making at most Q queries to the random oracle is therefore

$$\varepsilon_{\text{sound}} \leq (Q+1)\cdot 2^{-\rho\cdot(b-\log(n-1))}.$$

This means that the loss in soundness from the case of special soundness is $\rho\cdot\log(n-1)$ (i.e., from $\rho\cdot b$ to $\rho\cdot(b-\log(n-1))$). This loss can be compensated by either increasing ρ or b, in the same way as described above for special soundness. That is, for special soundness we choose ρ and b so that $\rho\cdot b \geq \kappa_c$. Here, the only change needed is to choose ρ and b so that $\rho\cdot(b-\log(n-1)) \geq \kappa_c$.

Computational cost. As above, let T_1 denote the computational cost of computing the first prover message in the Sigma protocol, and assume that the second prover message has insignificant computational cost. Then, we have that the expected cost of computing the Fischlin transform to the Sigma protocol with n-special soundness is $T = \rho\cdot T_1 + \rho\cdot 2^b\cdot T_{\text{hash}}$. Recall that for (regular) special soundness, the equation for the cost of generating a proof is exactly the same. However, this is misleading, since here we need $\rho\cdot(b-\log(n-1)) \geq \kappa_c$, whereas for regular special soundness we only needed $\rho\cdot b \geq \kappa_c$. To make this comparison clear, we can rewrite the equations only as a function of ρ:

- *Regular special soundness:* $T = \rho\cdot T_1 + \rho\cdot 2^{\frac{\kappa_c}{\rho}}\cdot T_{\text{hash}}$
- *n-special soundness:* $T = \rho\cdot T_1 + \rho\cdot 2^{\frac{\kappa_c}{\rho}+\log(n-1)}\cdot T_{\text{hash}} = \rho\cdot T_1 + \rho\cdot(n-1)\cdot 2^{\frac{\kappa_c}{\rho}}\cdot T_{\text{hash}}$

We therefore see that generating a proof with n-special soundness has n times the number of expected hash function operations, as expected, but the same amount of first prover message computations. Of course, this comparison holds when ρ is fixed; however, there is no requirement to choose the same value of ρ and as we have seen, the optimal ρ changes when there are more hash computations.

5.2 Batch Proofs

An application of specific interest for n-special soundness is batch proofs. For example, the batch Schnorr proof of [GLSY04] enables one to prove knowledge of n discrete logs close to the cost of proving knowledge of a single discrete log. However, extracting the discrete logs requires rewinding $n+1$ times to extract all n discrete logs, and so cannot be UC secure. We will demonstrate our method on this Sigma protocol, and remark that it is useful in some settings of MPC, like in Feldman VSS for a dishonest majority [CL24]. For the sake of concreteness, the proof of [GLSY04] is given below:

1. *Statement:* a set of group elements $Q_1,\ldots,Q_n$ with witnesses $w_1,\ldots,w_n$ (i.e., $Q_i = w_i\cdot G$ for all $i\in[n]$)
2. ProverFirstMessage: choose a random $r \in_R \mathbb{Z}_q$ and compute $R = r\cdot G$
3. ProverSecondMessage: compute $z \leftarrow r + \sum_{i=1}^{n} e^i\cdot w_i \bmod q$
4. VerifyProof: verify that $(\{Q_i\}_{i=1}^{n}, R)$ are group elements and $R = z\cdot G - \sum_{i=1}^{n} e^i\cdot Q_i$

Observe that the first prover message for this batch proof has the same cost as for the standard discrete log Sigma protocol for a single statement. This is what makes it so attractive from an efficiency perspective.

In order to see why our approach is significant, it is instructive to compare the cost of applying the Fischlin transform with $(n+1)$-special soundness to this batch proof to the alternative of simply running the basic discrete log proof n times. As we have mentioned, the cost of the first prover message for the batch case is the *same* as for a single proof. Furthermore, the batch proof for n statements has $(n+1)$-special soundness, since the extractor needs to obtain $n+1$ responses to interpolate the polynomial of degree-n defined by $p(x) = r + \sum_{i=1}^{n} w_i \cdot x^i$. We have the following costs:

- *n copies of a single proof:* the expected cost of proving is $T = n \cdot \rho \cdot T_1 + n \cdot \rho \cdot 2^{\frac{\kappa_c}{\rho}} \cdot T_{\text{hash}}$
- *A single batch proof:* the expected cost of proving is $T = \rho \cdot T_1 + \rho \cdot n \cdot 2^{\frac{\kappa_c}{\rho}} \cdot T_{\text{hash}}$

We conclude that for the *same* ρ, the cost of the single batch proof has the same number of hashes, but only ρ computations of T_1 versus $n \cdot \rho$ computations of T_1 when generating n copies of a single proof. Thus, this is clearly already much more efficient. Furthermore, as we will see below, different choices of ρ and b will be optimal here.

Analytic parameter optimization. Using the same method as in Section 4, we compute the optimal trade-off for the batch proof, for different values of n. We compute this only for a 2019 MacBook Pro with an Intel 2.3 GHz i9 CPU for which $a = 23$ for secp256k1 and $a = 13$ for Ed25519. The equation that we use here is $T = a \cdot \rho + \rho \cdot n \cdot 2^{\rho/\kappa_c}$.

Table 7: Analytically computed trade-offs for the batch discrete log on a 2019 MacBook Pro with an Intel 2.3 GHz i9 CPU. Recall that $t = b + 5$ for all $\rho \leq 64$, and the soundness is $\rho \cdot (b - \log n)$.

Protocol	$n = 16$	$n = 128$	$n = 256$
secp256k1	$\rho_{\min} = 64.9$: $\rho = 64, b = 6$	$\rho_{\min} = 83.5$: $\rho = 64, b = 9$	$\rho_{\min} = 86$: $\rho = 64, b = 10$
Ed25519	$\rho_{\min} = 71.8$: $\rho = 64, b = 6$	$\rho_{\min} = 85.6$: $\rho = 64, b = 9$	$\rho_{\min} = 87.1$: $\rho = 64, b = 10$

In contrast to the case of a single proof., there is no difference between the parameters for secp256k1 and Ed25519 here According to our experiments, for our final best method using all our optimizations shown below, we have that we should take $\rho = 64$ for all values of $n \geq 8$ for secp256k1, and for $n \geq 5$ for Ed25519; below that, we take $\rho = 43$. We therefore have the following parameter recommendations (for the case of $a = 23$). For **secp256k1**, take $[\rho = 43,\ b = \log n + 3,\ t = b + 5]$ for $2 \leq n < 8$, and take $[\rho = 64,\ b = \log n + 2,\ t = b + 5]$ for $n \geq 8$. For **Ed25519**, take $[\rho = 43,\ b = \log n + 3,\ t = b + 5]$ for $2 \leq n < 5$, and take $[\rho = 64,\ b = \log n + 2,\ t = b + 5]$ for $n \geq 5$.

5.3 Optimizing the Batch Proof

Unfortunately, a naive implementation of the batch proof of [GLSY04] described above turns out to be extremely inefficient, and much worse than just repeating the basic proof. The main reason is that the cost of computing $z = r + \sum_{i=1}^{n} e^i \cdot w_i \bmod q$ is not at all insignificant! In our optimization for the single-proof discrete log described in Section 2.3, we showed that each iteration requires a single modular addition only. In contrast, here we need to compute a degree-n polynomial which is much more expensive than hashing. In this section, we describe implementation optimizations that we carried out in order to achieve much faster running times.

Reducing the number of multiplications and additions. The naive approach is to simply compute $z_i = r_i + \sum_{j=1}^{n} {e_i}^j \cdot w_j \bmod q$ in ProverSecondMessage in every iteration. Using Horner's algorithm for polynomial evaluation, this costs n modular multiplications and additions in *every* iteration, and so the expected number of these operations is $n \cdot \rho \cdot 2^b$.

The first thing to notice is that the sum $\sum_{j=1}^{n} e^j \cdot w_j \bmod q$ is the same for all $i \in [\rho]$ (i.e., it is the same in each of the ρ proofs generated, and only the r_i value changes). Thus, we can compute $\sum_{j=1}^{n} {e_i}^j \cdot w_j \bmod q$ once only for all $e_i = 0, \ldots, T$ where $T = \max_{i \in [\rho]}(e_i)$ (in the worst case, $T = 2^t - 1$, but we expect it to be lower), and then merely add r_i in order to obtain z_i within the internal loop. This reduces the cost to $n \cdot T$ modular multiplications and additions, instead of $n \cdot \rho \cdot 2^b$. We remark that since T is not known ahead of time, the computation of each $\sum_{j=1}^{n} {e_i}^j \cdot w_j \bmod q$ should be computed on-demand the first time than an e_i is needed, and then just retrieved from the array each time after that. This yields the following algorithm, written as a single flow:

Prover:

- *Input:* Statements $Q_1, \ldots, Q_n$ and witnesses $w_1, \ldots, w_n$
- *Parameters:* ρ, b such that $\rho \cdot (b - \log n) \geq \kappa_c$ and $t \in \{b+5, b+6\}$ as described
- *The algorithm:*

```
1. For $i = 1, \ldots, \rho$,
   (a) $r_i \in_R Z_q$
   (b) $R_i \leftarrow r_i \cdot G$
2. For $e_i = 0, \ldots, 2^t - 1$
   (a) $\mathsf{poly}[e_i] = \sum_{j=1}^{n} {e_i}^j \cdot w_j \bmod q$
   This is computed using Horner's algorithm, and is best computed on-demand
   inside the internal loop.
3. Let $\vec{R} = (R_1, \ldots, R_\rho)$
4. $\mathsf{common\text{-}h} \leftarrow \mathcal{H}(Q_1, \ldots, Q_n, \vec{R}, \mathsf{sid})$
5. For $i = 1, \ldots, \rho$:
   (a) For $e_i = 0, \ldots, 2^t - 1$
       i.   $z_i \leftarrow r_i + \mathsf{poly}[e_i] \bmod q$
       ii.  $h_i \leftarrow \mathcal{H}_b(\mathsf{common\text{-}h}, i, e_i, z_i)$
       iii. If $h_i = 0$, break
       iv.  If $e_i = 2^t - 1$, redo the proof from the beginning
            (If this occurs, then it means that no break ever took place, meaning that the proof
            failed.)
6. $\vec{e} \leftarrow (e_1, \ldots, e_\rho)$
7. $\vec{z} \leftarrow (z_1, \ldots, z_\rho)$
8. $\pi \leftarrow \left(\vec{R}, \vec{e}, \vec{z}, b, \rho\right)$
9. Output $\pi$
```

The cost of the internal loop is just a single modular addition, like in the optimization for a single discrete log proof. However, the cost of computing the poly array is still significant.

Verifier:

1. Parse π as $\left(\vec{R}, \vec{e}, \vec{z}, b, \rho\right)$
2. If the size of $\vec{R}$, $\vec{e}$, and $\vec{z}$ are not ρ, then output reject
3. If $b \cdot (\rho - \log n) < \kappa_c$, then output reject
4. Verify that all $Q_1, \ldots, Q_n, R_1, \ldots, R_\rho$ are on the curve (in the subgroup) and non-zero
5. $\mathsf{common\text{-}h} \leftarrow \mathcal{H}(x, \vec{R}, \mathsf{sid})$
6. For $i \in \{1, \ldots, \rho\}$
   (a) Halt and output reject if $R_i \neq z_i \cdot G - \sum_{j=1}^{n} {e_i}^j \cdot Q_j$
   (The sum $\sum_{j=1}^{n} {e_i}^j \cdot Q_j$ must also be computed using Horner's algorithm.)
   (b) Halt and output reject if $\mathcal{H}_b(\mathsf{common\text{-}h}, i, e_i, z_i) \neq 0$
7. Output accept

Additional optimizations. In order to further optimize the implementation, we observe the following two facts. First, the most computationally intense part of the prover's algorithm is to evaluate the polynomial $p(x) = \sum_{j=0}^{n-1} w_j \cdot x^j$ over many values of x (these are the e_i values). Thus, our optimizations should focus on this step. Second, the Fischlin transform actually does not care what values of e are used. It is written for $e = 0, \ldots, 2^t - 1$, but in actuality any fixed set of 2^t points would work in exactly the same way. This means that we can use faster methods for polynomial evaluation that rely on specific points. Below, we will explore options for this.

Using one step of FFT to further reduce the cost of computing poly. As observed above, we actually don't care at all about which set of e_i values are used, as long as there is a large enough set. As such, one could use n'th roots of unity and FFT to compute these values. However, this requires having n'th roots of unity of the right size, which is not always the case. However, we can *always* use the *first step* of FFT with e_i and $-e_i$ in order to lower the cost by a half.

In order to see how this works, let $p(x) = \sum_{i=1}^{n} w_i \cdot x^i \bmod q$ and define $p_0(x) = \sum_{i=1}^{n/2} w_{2\cdot i} \cdot x^i \bmod q$ and $p_1(x) = \sum_{i=0}^{n/2-1} w_{2\cdot i+1} \cdot x^i \bmod q$. Then it holds that $p(x) = p_0(x^2) + x \cdot p_1(x^2) \bmod q$, implying that for any $a \in \mathbb{Z}_q$, it holds that $p(a) = p_0(a^2) + a \cdot p_1(a^2)$ and $p(-a) = p_0(a^2) - a \cdot p_1(a^2)$. This yields the following algorithm:

1. For $e_i = 0, \ldots, 2^{t-1}$
   (a) Compute $\alpha \leftarrow p_0({e_i}^2) \bmod q$ using Horner
   (b) Compute $\beta \leftarrow p_1({e_i}^2) \bmod q$ using Horner
   (c) Output $\mathsf{poly}[e_i] \leftarrow \alpha + e_i \cdot \beta \bmod q$
   (d) Output $\mathsf{poly}[-e_i] \leftarrow \alpha - e_i \cdot \beta \bmod q$ (note that $-e_i = q - e_i$)

The cost of computing 2^t values of e_i in $\{q - 2^{t-1}, q - 2^{t-1} + 1, \ldots, 0, \ldots, 2^{t-1} - 1, 2^{t-1}\}$ therefore becomes $n \cdot 2^{t-1}$ instead of $n \cdot 2^t$, which is half the cost.

Using additional steps of FFT. It is well known that there exists an n'th root of unity in a field of size q if and only if n divides $q-1$. In such a case, we can run FFT to compute n points, rather than just 2 (the above method of using one step of FFT is equivalent to saying that there is a 2'th root of unity, which is always true). For secp256r1, we have that 2^4 divides $q-1$; for secp256k1 we have that 2^6 divides $q-1$, and for Ed25519 we have that 2^2 divides $q-1$. As such for these curves, we can run the above method at depths 4, 6 and 2, respectively. In order to see how this "partial" FFT works, observe that all is needed is to halt the recursion after the number of supported steps, and to then call Horner to compute the polynomial on the remaining number of points.

Using finite differences to reduce the cost of computing poly. Another optimization idea is to use finite difference of polynomials to avoid multiplications. Here, we introduce a *novel* polynomial evaluation technique that works on sequential values. First, we introduce the *difference operator* Δ notation.

Definition 3. Given a function f, The first order *(forward) difference* of f is

$$\Delta^1\left[f\right](x) \stackrel{\text{def}}{=} f(x+1) - f(x).$$

A higher order difference of f can be recursively defined as

$$\Delta^{i+1}\left[f\right](x) \stackrel{\text{def}}{=} \Delta^1\left[\Delta^i\left[f\right]\right](x) = \Delta^i\left[f\right](x+1) - \Delta^i\left[f\right](x) \text{ for } i \in \mathbb{N}.$$

When the order is 1, it can be omitted and written as $\Delta\left[f\right](x)$.

In order to make it clearer as to what the forward difference is, observe that

$$\Delta^2\left[f\right](x) = \Delta^1\left[f\right](x+1) - \Delta^1\left[f\right](x) = f(x+2) - 2\cdot f(x+1) + f(x), \text{ and}$$

$$\Delta^3\left[f\right](x) = \Delta^2\left[f\right](x+1) - \Delta^2\left[f\right](x) = f(x+3) - 3\cdot f(x+2) + 3\cdot f(x+1) - f(x).$$

Linearity and commutativity. Note that the operator Δ is $\mathsf{commutative}$, and so $\Delta^{i+1}\left[f\right](x) = \Delta^i\left[\Delta\left[f\right]\right](x)$. In order to see this, we first observe that the operator is linear, in the sense that: $\Delta^i\left[f\right](x) + \Delta^i\left[g\right](x) = \Delta^i\left[f+g\right](x)$. We prove this by induction.

The base case is immediate since

$$\Delta^1\left[f\right](x) + \Delta^1\left[g\right](x) = f(x+1)+g(x+1)-f(x)-g(x) = (f+g)(x+1)-(f+g)(x) = \Delta^1\left[f+g\right](x).$$

Assume this holds for k, and we prove for $k+1$. We have:

$$\begin{aligned}
\Delta^{k+1}\left[f\right](x) + \Delta^{k+1}\left[g\right](x) &= \Delta^k\left[f\right](x+1) - \Delta^k\left[f\right](x) + \Delta^k\left[g\right](x+1) - \Delta^k\left[g\right](x)\\
&= \big(\Delta^k\left[f\right](x+1) + \Delta^k\left[g\right](x+1)\big) - \big(\Delta^k\left[f\right](x) + \Delta^k\left[g\right](x)\big)\\
&= \Delta^k\left[f+g\right](x+1) - \Delta^k\left[f+g\right](x)\\
&= \Delta^{k+1}\left[f+g\right](x)
\end{aligned}$$

where the third equality is from the assumption of the induction regarding k. This implies commutativity since

$$\Delta^{i+1}\left[f\right](x) = \Delta^i\left[f\right](x+1) - \Delta^i\left[f\right](x) = \Delta^i\left[f(x+1) - f(x)\right] = \Delta^i\left[\Delta\left[f\right]\right](x)$$

where the second equality follows by writing $g(x) = f(x+1)$ in which case $\Delta^i\left[f(x+1)\right] = \Delta^i\left[g\right](x))$ and so the equality follows from linearity.

Finite difference property of polynomials. We prove the following proposition which is central to our new algorithm in Appendix B.

Proposition 1. *Let $f(x)$ be an n-degree polynomial with a being the coefficient of x^n. Then $\Delta\left[f\right](x)$ is a degree-$(n-1)$ polynomial and $\Delta^n\left[f\right](x) = a\cdot n!$ is a constant.*

The algorithm – background. Given the polynomial $f(x) = w_n \cdot x^n + \cdots + w_1 \cdot x + w_0$, to visualize how we can evaluate on 2^t positions, we put $\Delta^k [f] (i)$ for $k \in [n] \cup \{0\}$ and $i \in \mathbb{Z}_q$ into a matrix:

$$\begin{bmatrix} \vdots & \vdots & \cdots & \vdots & & & \\ f(i) & \Delta^1 [f] (i) & \cdots & \Delta^k [f] (i) & \Delta^{k+1} [f] (i) & \cdots & \Delta^n [f] (i) \\ f(i+1) & \Delta^1 [f] (i+1) & \cdots & \Delta^k [f] (i+1) & \Delta^{k+1} [f] (i+1) & \cdots & \Delta^n [f] (i+1) \\ \vdots & \vdots & \cdots & \vdots & & & \end{bmatrix}.$$

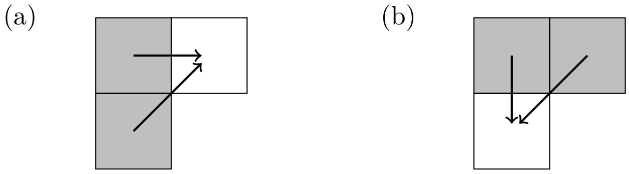
Our goal is to calculate 2^t elements in the left most column in order to obtain 2^t values $f(i)$. By Proposition 1, we know that $\Delta^n [f] (i) = w_n \cdot n!$ for all $i \in \mathbb{Z}_q$. This implies that the entire rightmost column all have the same value $w_n \cdot n!$. Thus, once we can compute a single value in this column, the rest are obtained by just copying.

Next we show how it is possible to compute values in the matrix from others. By Definition 3:

(a) $\Delta^{k+1} [f] (i) = \Delta^k [f] (i+1) - \Delta^k [f] (i)$ and

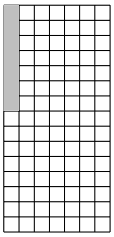
(b) $\Delta^k [f] (i+1) = \Delta^k [f] (i) + \Delta^{k+1} [f] (i)$ (this follows from (a) above)

If we visualize these operations in the matrix, we have that knowing the values of two grays cells at the appropriate relative positions, we can calculate the white ones using a *single addition operation* as illustrated below:

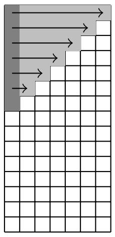


Given the above operations, we can construct an algorithm to efficiently compute $f(x)$ on T consecutive values $x = a, \ldots, a + T - 1$ for some a. The idea is to begin with n computations of the polynomial f, and then use those values to efficiently fill out the rest of the matrix using those values.

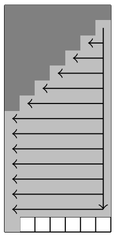
We also use the matrix diagram to visualize the steps, where light-gray cells represent the ones that are being calculated in the current step and dark-gray cells represent the ones that were already calculated. The three phases are depicted as follows:



Phase 1 of the algorithm



Phase 2 of the algorithm



Phase 3 of the algorithm

The algorithm. We proceed to describe each of the three phases:

1. **Calculate $n+1$ consecutive evaluations in the top of the left column of the matrix:** We begin by computing $n+1$ values $f(a), f(a+1), \ldots, f(a+n)$.
2. **Fill out the upper left triangle of the matrix:** We use the operation (a) above to fill out the values as shown in the diagram below. The columns of the matrix are denoted i and the rows are denoted j.

```
• For $i = 1, \dots, n$
    For $j = a + n - i, \dots, a$
      $\Delta^i [f] (j) \leftarrow \Delta^{i-1} [f] (j + 1) - \Delta^{i-1} [f] (j)$
```

3. **Compute the rest of the matrix, including the rest of the first column:** We use operation (b) above in order to work back from the upper left triangle, and fill out the rest of matrix, which includes the entire first column (which contains the values that we want).

```
• $\Delta^n [f] (a + 1) \leftarrow \Delta^n [f] (a)$
• For $j = a + 2, \dots, a + T - 2$
    – $\Delta^n [f] (j) \leftarrow \Delta^n [f] (j - 1)$
    – For $i = n - 1$ down to 0
        If $\Delta^i [f] (j)$ is not calculated,[8] $\Delta^i [f] (j) \leftarrow \Delta^i [f] (j - 1) + \Delta^{i+1} [f] (j - 1)$
• $f(a + T - 1) = f(a + T - 2) + \Delta^1 [f] (a + T - 2)$.
```

In Step 1, we did not specify how exactly to compute $f(a), f(a + 1), \dots, f(a + n)$. One way is to set $a = -n/2$, then we can use "one-step FFT" and Horner's rule, which gives us an algorithm with $n^2/2$ addition and multiplication. Note that we cannot apply additional steps of FFT since we require consecutive evaluation points.

Cost: In summary, the algorithm requires $\frac{n^2}{2}$ multiplications and additions for Step 1, and $\sim n \cdot 2^t$ additions (to fill out the rest of the matrix of size 2^t). In comparison, the one-step FFT approach has complexity $n \cdot 2^{t-1}$ multiplications and additions. The cost ratio between these methods depends on the cost of multiplication versus addition. For example, for 256-bit modular additions and multiplications, on MacBook Pro 2019 with an Intel 2.3 GHz i9 CPU the cost of multiplication is 2.4 times the cost of addition. As such, we have that the one-step FFT approach would cost $3.4 \cdot n \cdot 2^{t-1}$ additions, whereas the finite differences approach would cost $3.4 \cdot \frac{n^2}{2} + n \cdot 2^t$ additions. In this case, the finite differences approach is faster for $n < \frac{0.7}{1.7} \cdot 2^t = 0.412 \cdot 2^t$. As we have shown above, for this batch proof, we always take either $b = \log n + 3$ or $b = \log n + 2$ and $t = b + 5$. Thus, we have that $t = \log n + 7$ or $t = \log n + 8$ always, and so $0.412 \cdot 2^t$ is always much bigger than n. To be concrete, consider the case of $n = 128$ with $t = 14$ (as in Table 7). With the finite differences approach, the approximate cost is the equivalent of 2.125 million additions. In contrast, the cost of the one-step FFT approach is 3.57 million additions. For another example, for the case of $n = 16$ with $t = 12$, the approximate cost is the equivalent of 65,971 additions. In contrast, the cost of the one-step FFT approach is 111,411 additions.

Implementation details. Since we wish to have a complexity that depends on the expected cost $\rho \cdot 2^b$ and not 2^t, we can compute this matrix on-demand, by first filling out the first n rows, and then computing additional rows (at the cost of n additions) only when a new value is needed. In addition, when working this way, we only need to allocate an initial $n \times n$ matrix, and the rest can be computed using only the previous and current row. This can be significant when 2^t is large.

A different look of the finite differences method. The finite differences method is actually an algorithm that receives any d sequential points $p(a), p(a + 1), \dots, p(a + d)$ on a degree-d polynomial $p(x)$, and after carrying out an initial d^2 additions can compute additional points $p(a + d + 1), p(a + d + 2), \dots$ each one with an additional cost of just d *additions*. Equivalently, after receiving d sequential points, the algorithm computes the next $N - d$ sequential points at the cost of just $N \cdot d$ additions.

[8] In the range of $j \in \{-n/2, \dots, n/2\}$ half of the matrix has already been computed, so we don't want to compute it again.

5.4 Experimental Results for the Batch Discrete Log Proof

In this section, we present experimental results for the batch discrete log proof, comparing the following methods for different values of n:

- **Repeat:** this simply involves running the basic discrete log proof separately for each statement; this is the baseline
- **Horner:** this involves computing each polynomial using Horner's method only (but computing the values once only on-demand)
- **Horner+1-FFT:** this involves running 1-step of FFT and using Horner to compute the rest of the polynomial
- **FFT:** this involves running as many steps as possible of FFT and then using Horner
- **Final:** this is our final implementation that uses the finite difference method, together with Horner and one step of FFT in order to compute the first n points

Experimental results. We provide graphs below for secp256k1 and Ed25519. Times are in milliseconds, and are the average over 1,000 executions. In each figure the graph on the left-hand side shows running-times for batch sizes of up to 32, and the graph on the right-hand side shows for batch sizes of up to 256 (the graph on the left shows the first 32 in more detail).

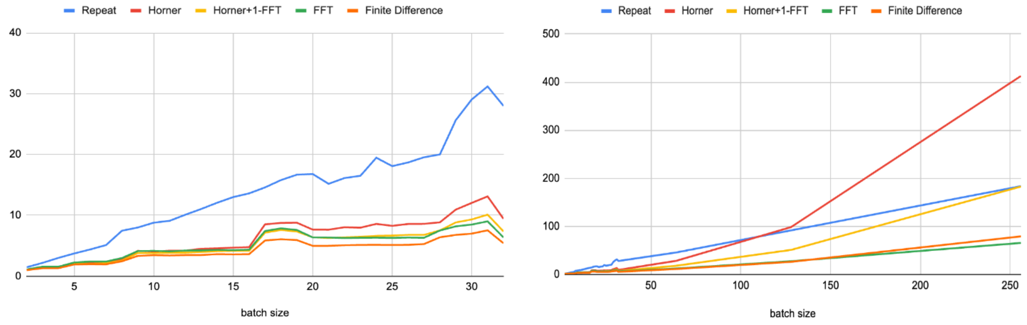


Figure 1: Batch Discrete Log for secp256k1

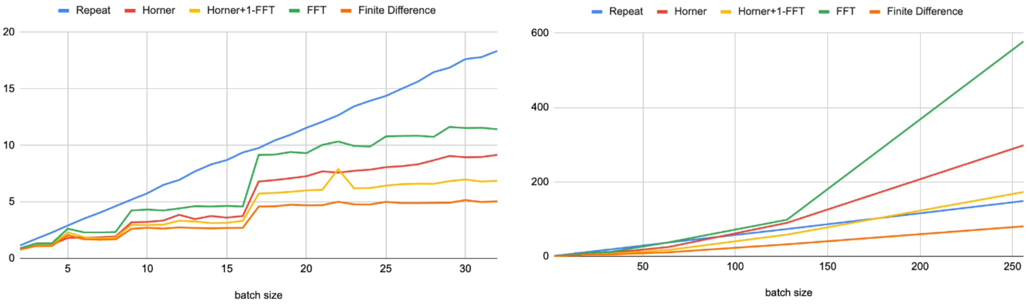


Figure 2: Batch Discrete Log for Ed25519

The results are very interesting. For secp256k1, we can see that for most parameters our finite difference method outperforms all others. Furthermore, the improvements over naive repetition are striking. For example, for secp256k1 and $n = 16$ we have 13.6ms for repetition and 3.61ms for finite differences (3.8 times faster), for $n = 32$ we have 28ms for repetition and 5.4ms for finite differences (5.2 times faster). The improvements are similar for Ed25519. It is also worth noting that the finite difference method outperforms Horner with one step of FFT, running about 1.25 times faster for n's in the range of 16-32, and running about twice as fast for larger n's.

Interestingly, Horner becomes worse than plain repetition for large batches (the polynomial evaluation becomes more expensive). Another interesting result is that for large batches FFT performs worse than Horner for Ed25519. This is because FFT for Ed25519 only has depth 2, but requires using FFT values of e_i which are large. These larger values make all operations more expensive.

Finally, observe that the running times (apart from repetition) look a bit like a step function, increasing at $n = 4, 8, 16$. This is primarily because we need to increase b by $\lceil \log n \rceil$ for soundness, and so there are jumps at the powers of 2.

Proof size. We conclude by remarking that the proof size of the batch proof is much smaller than repeating multiple times. Indeed, the size of the batch proof is the *same size as an individual proof*, and thus batching n proofs together reduces the proof size by a factor of n. This can be very significant.

Acknowledgements

We thank Peihan Miao for helpful comments. The second author also thanks Marc Fischlin [Fis05] for fond memories of bike riding around Santa Barbara on the Tuesday free afternoon at CRYPTO in the early 2000s.

References

[BGR98] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*, pages 236–250. Springer, 1998. URL: https://doi.org/10.1007/BFb0054130, doi:10.1007/BFB0054130.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001. doi:10.1109/SFCS.2001.959888.

[CDS94] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994. doi:10.1007/3-540-48658-5_19.

[CGKN21] Konstantinos Chalkias, François Garillot, Yashvanth Kondi, and Valeria Nikolaenko. Non-interactive half-aggregation of eddsa and variants of schnorr signatures. In Kenneth G. Paterson, editor, *Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17-20, 2021, Proceedings*, volume 12704 of *Lecture Notes in Computer Science*, pages 577–608. Springer, 2021. doi:10.1007/978-3-030-75539-3_24.

[CL24] Yi-Hsiu Chen and Yehuda Lindell. Feldman's verifiable secret sharing for a dishonest majority. *IACR Cryptol. ePrint Arch.*, page 31, 2024. URL: https://eprint.iacr.org/2024/031.

[Dam10] Ivan Damgård. On sigma protocols., 2010. URL: https://www.cs.au.dk/~ivan/Sigma.pdf.

[DV14] Özgür Dagdelen and Daniele Venturi. A second look at fischlin's transformation. In David Pointcheval and Damien Vergnaud, editors, *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, volume 8469 of *Lecture Notes in Computer Science*, pages 356–376. Springer, 2014. doi:10.1007/978-3-319-06734-6_22.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *Lecture Notes in Computer Science*, pages 152–168. Springer, 2005. doi:10.1007/11535218_10.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986. doi:10.1007/3-540-47721-7_12.

[GLSY04] Rosario Gennaro, Darren Leigh, Ravi Sundaram, and William S. Yerazunis. Batching schnorr identification scheme with applications to privacy-preserving authorization and low-bandwidth communication devices. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 276–292. Springer, 2004. doi:10.1007/978-3-540-30539-2_20.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols - Techniques and Constructions.* Information Security and Cryptography. Springer, 2010. doi:10.1007/978-3-642-14303-8.

[HLNR18] Iftach Haitner, Yehuda Lindell, Ariel Nof, and Samuel Ranellucci. Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody. Cryptology ePrint Archive, Paper 2018/987, 2018. https://eprint.iacr.org/2018/987. URL: https://eprint.iacr.org/2018/987, doi:10.1145/3243734.3243788.

[KS22] Yashvanth Kondi and Abhi Shelat. Improved straight-line extraction in the random oracle model with applications to signature aggregation. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part II*, volume 13792 of *Lecture Notes in Computer Science*, pages 279–309. Springer, 2022. doi:10.1007/978-3-031-22966-4_10.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96,* *International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 387–398. Springer, 1996. doi:10.1007/3-540-68339-9_33.

A Proof of Batch Verification

The proof of correctness and security of the batch verification method of Section 2.3 is as follows. *Correctness* follows from the fact that

$$R_{\text{sum}} = \sum_{i=1}^{\rho} \sigma_i \cdot R_i = \sum_{i=1}^{\rho} \sigma_i \cdot (z_i \cdot G - e_i \cdot Q) = \sum_{i=1}^{\rho} (\sigma_i \cdot z_i) \cdot G - \sum_{i=1}^{\rho} (\sigma_i \cdot e_i) \cdot Q = z_{\text{sum}} \cdot G - e_{\text{sum}} \cdot Q.$$

Security follows from the fact that if there exists a $j \in [\rho]$ for which $R_j \neq z_j \cdot G - e_j \cdot Q$, then it follows that $R_{\text{sum}} = z_{\text{sum}} \cdot G - e_{\text{sum}} \cdot Q$ if and only if

$$\sum_{i=1}^{\rho} \sigma_i \cdot R_i = \left(\sum_{i=1}^{\rho} \sigma_i \cdot z_i \right) \cdot G - \left(\sum_{i=1}^{\rho} \sigma_i \cdot e_i \right) \cdot Q$$

which holds if and only if

$$\sigma_j \cdot R_j + \left(\sum_{i \neq j} \sigma_i \cdot R_i \right) = \sigma_j \cdot z_j \cdot G - \sigma_j \cdot e_j \cdot Q + \left(\sum_{i \neq j} \sigma_i \cdot z_i \right) \cdot G - \left(\sum_{i \neq j} \sigma_i \cdot e_i \right) \cdot Q$$

which in turn holds if and only if

$$\sigma_j \cdot (R_j - (z_j \cdot G - e_j \cdot Q)) = \left(\sum_{i \neq j} \sigma_i \cdot z_i \right) \cdot G - \left(\sum_{i \neq j} \sigma_i \cdot e_i \right) \cdot Q - \left(\sum_{i \neq j} \sigma_i \cdot R_i \right).$$

Note that $\sigma_1, \ldots, \sigma_\rho$ are all chosen after all the other values are fixed. Consider now the mental experiment where all σ_i with $i \neq j$ are chosen first, and then σ_j is chosen last. This means that the right-hand side is a fixed group element (which could be zero or any other point). Furthermore, $R_j \neq z_j \cdot G - e_j \cdot Q$, and so $R_j - (z_j \cdot G - e_j \cdot Q) \neq 0$. Since we are dealing with prime-order groups, every non-zero element is a generator. Thus, the set $\{\sigma_j \cdot (R_j - (z_j \cdot G - e_j \cdot Q))\}_{\sigma_j \in \{0,1\}^{\kappa_s}}$ is of size 2^{κ_s}. Since the value on the right-hand side is fixed before σ_j is chosen, and since σ_j is chosen uniformly, this implies that equality holds with probability at most $2^{-\kappa_s}$, as required.[9]

B Proof of Proposition 1

Proof. We prove the proposition by induction. Let $p(x)$ be a degree-n polynomial such that the coefficient of x^n is a. For the base case of $n = 1$ case, we can write $p(x) = a \cdot x + b$, and we have:

$$\Delta^1 [p] (x) = p(x+1) - p(x) = a \cdot (x+1) + b - ax - b = a$$

Thus, $\Delta [f] (x)$ is an $(n-1) = 0$-degree polynomial and $\Delta^n [f] (x) = \Delta^1 [p] (x) = a = a \cdot n!$, as required.

[9] This is a well-known technique, but the proof has been included for the sake of completeness.

Next, assume that the theorem holds for $n = k$ and so for any degree-k polynomial $p(x)$ it holds that $\Delta\left[p\right](x)$ is a degree-$(k-1)$ polynomial and $\Delta^k\left[p\right](x) = a \cdot k!$. We prove that it holds for $n = k+1$. Let $p(x) = a \cdot x^{k+1} + g(x)$ where g is a k-degree polynomial. Then

$$\begin{aligned}
\Delta\left[p\right](x) &= p(x+1) - p(x)\\
&= a \cdot (x+1)^{k+1} + g(x+1) - a \cdot x^{k+1} - g(x)\\
&= a \cdot \left((x+1)^{k+1} - x^{k+1}\right) + \Delta\left[g\right](x)\\
&= a \cdot \left(\sum_{i=0}^{k+1} \binom{k+1}{i} \cdot x^i - x^{k+1}\right) + \Delta\left[g\right](x)\\
&= a \cdot \sum_{i=0}^{k} \binom{k+1}{i} \cdot x^i + \Delta\left[g\right](x)\\
&= a \cdot (k+1) \cdot x^k + h(x)
\end{aligned}$$

for some polynomial $h(x)$ (the fourth equality is from $(1+x)^n = \sum_{i=0}^{n} \binom{n}{i} \cdot 1^{n-1} \cdot x^i$). By the assumption, we have that $\Delta\left[g\right](x)$ is a degree-$(k-1)$ polynomial, and thus $h(x)$ is a degree-$(k-1)$ polynomial (the highest order in the sum is x^k and the rest are smaller). Thus, we have that $\Delta\left[p\right](x)$ is a degree-k polynomial, as required.

Furthermore, since we have already established that $h(x)$ is a degree-$(k-1)$ polynomial, the coefficient of x^k in the degree-k polynomial $\Delta\left[p\right](x)$ is $a \cdot (k+1)$. We have

$$\Delta^{k+1}\left[p\right](x) = \Delta^k\left[\Delta\left[p\right]\right](x) = \Delta^k\left[a \cdot (k+1) \cdot x^k\right](x) + \Delta^k\left[h\right](x) = a \cdot (k+1) \cdot k! + 0 = a \cdot (k+1)!$$

where the first equality is by commutativity, the second by linearity (and the identity proven above for $\Delta\left[p\right](x)$), and the third equality is by the induction assumption (and that h is of degree-$k-1$). This completes the proof. □

C Additional Examples

In this section, we consider two more Sigma protocol examples. Specifically, we look at two proofs of knowledge needed in the multiparty ECDSA protocol of [HLNR18]: the first is called ZKElGamalCommit and is a proof of knowledge of an ElGamal commitment value, and the second is called ZKElGamalCommit-Mult-Private-Scalar, and it is a proof of knowledge that an ElGamal commitment has been multiplied by a private scalar and rerandomized.

Formally, let Q be a random group element. Then an ElGamal commitment to a value x is the pair $(A, B) = (r \cdot G, r \cdot Q + x \cdot G)$; such commitments are useful since they are additively homomorphic. The relations for the two proofs we consider are:

$$\mathcal{R}_{\mathsf{ElGamalCommit}} = \left\{\left((G, Q, A, B), (x, r)\right) \mid A = r \cdot G \text{ and } B = r \cdot Q + x \cdot G\right\}$$

and

$$\begin{aligned}
&\mathcal{R}_{\mathsf{ElGamalCommit\text{-}Mult\text{-}Private\text{-}Scalar}}\\
&\quad = \left\{\left((G, Q, A_1, B_1, A_2, B_2), (c, r)\right) \mid A_2 = c \cdot A_1 + r \cdot G \text{ and } B_2 = c \cdot B_1 + r \cdot Q\right\}.
\end{aligned}$$

Observe that if $(A_1, B_1) = (r_1 \cdot G, r_1 \cdot Q + x \cdot G)$ is a commitment to x, then $A_2 = (c \cdot r_1 + r) \cdot G$ and $B_2 = (c \cdot r_1 + r) \cdot Q + c \cdot x \cdot G$ and thus (B_1, B_2) is an independent commitment to $c \cdot x$, using rerandomization r.

Sigma protocols. The Sigma protocols for these two relations (see [HLNR18, Sections A.1.2 and A.1.4]) are as follows:

- **ZK-ElGamalCommit:**
  1. *Statement:* a group element Q, a pair of elements (A, B) with witness (x, r) such that $A = r \cdot G$ and $B = r \cdot Q + x \cdot G$
  2. ProverFirstMessage: choose random $r', x' \in_R \mathbb{Z}_q$ and compute $A' = r' \cdot G$ and $B' = r' \cdot Q + x' \cdot G$
  3. ProverSecondMessage: compute $z_x = x' + e \cdot x \bmod q$ and $z_r = r' + e \cdot r \bmod q$
  4. VerifyProof: verify that Q, A, B, A', B' are group elements, and that $A' = z_r \cdot G - e \cdot A$ and $B' = z_r \cdot Q + z_x \cdot G - e \cdot B$
- **ZK-ElGamalCommit-Mult-Private-Scalar:**
  1. *Statement:* a group element Q, two pairs of elements (A_1, B_1) and (A_2, B_2) with witness (c, r)
  2. ProverFirstMessage: choose random $c', r' \in_R \mathbb{Z}_q$ and compute $A' = c' \cdot A_1 + r' \cdot G$ and $B' = c' \cdot B_1 + r' \cdot Q$
  3. ProverSecondMessage: compute $z_c = c' + e \cdot c \bmod q$ and $z_r = r' + e \cdot r \bmod q$
  4. VerifyProof: verify that $Q, A_1, B_1, A_2, B_2, A', B'$ are group elements, and

$$\begin{cases} A' = z_c \cdot A_1 + z_r \cdot G - e \cdot A_2 \\ B' = z_c \cdot B_1 + z_r \cdot Q - e \cdot B_2 \end{cases}$$

These Sigma protocols are considerably more expensive than the simple proof of knowledge of discrete log. In particular, the prover first message for discrete log is a single multiplication of the generator point, whereas for **ElGamalCommit** the first message has two multiplications of the generator and one of a random point, and for **ElGamalCommit-Mult-Private-Scalar** the first message has one multiplication of the generator and three of random points.

Analytically finding the parameters. We begin by analytically finding the optimal parameters for the different computing environments above for these two Sigma protocols. We denote by MUL-G the operation of multiplying the generator point by a scalar, and by MULT the operation of multiplying an arbitrary point by a scalar. We remark that the former operation is typically much cheaper since precomputation is used for the generator. We compute these parameters for the secp256k1 curve only.

The first step to computing these parameters is to find the ratio between MUL-G and MULT for each platform (we already have the ratio between MUL-G and SHA256 , and so this suffices). Once we have this ration, we can compute the equations. However, note that using Shamir's trick, the computation of $a \cdot G + b \cdot Q$ is about 58% the cost of $a \cdot G$ and $b \cdot Q$ separately.

- **Intel:** MULT is 2.3 times the cost of MUL-G . Recall that MUL-G is 42 times T_{hash} on this platform. This yields the following equations:
  1. ElGamalCommit:

$$\begin{aligned} T &= (\mathsf{MUL\text{-}G} + 0.58 \cdot (\mathsf{MUL\text{-}G} + \mathsf{MULT}) \cdot \rho + \rho \cdot 2^{128/\rho} \cdot T_{\text{hash}} \\ &= \left((42 + 0.58 \cdot (42 + 2.3 \cdot 42) \cdot \rho + \rho \cdot 2^{128/\rho}\right) \cdot T_{\text{hash}} \\ &= \left(122.4 \cdot \rho + \rho \cdot 2^{128/\rho}\right) \cdot T_{\text{hash}} \end{aligned}$$

2. $\mathsf{ElGamalCommitMultScalar}$:

$$\begin{aligned} T &= (2 \cdot \mathsf{MULT} + 0.58 \cdot (\mathsf{MUL\text{-}G} + \mathsf{MULT}) \cdot \rho + \rho \cdot 2^{128/\rho} \cdot T_{\text{hash}} \\ &= \left((2 \cdot 2.3 \cdot 42 + 0.58 \cdot (42 + 2.3 \cdot 42) \cdot \rho + \rho \cdot 2^{128/\rho}\right) \cdot T_{\text{hash}} \\ &= \left(273.6 \cdot \rho + \rho \cdot 2^{128/\rho}\right) \cdot T_{\text{hash}} \end{aligned}$$

- **WASM:** MULT is the same cost as $\mathsf{MUL\text{-}G}$, and $\mathsf{MUL\text{-}G}$ is 171 times the cost of SHA256. In the same way as above, we obtain:

1. $\mathsf{ElGamalCommit}$: $T = 369.4 \cdot \rho + \rho \cdot 2^{128/\rho}$

2. $\mathsf{ElGamalCommitMultScalar}$: $T = 540.4 \cdot \rho + \rho \cdot 2^{128/\rho}$

We remark that the same batch verification technique for discrete log can be used for verification in both of the above proofs. Using WolframAlpha, we have the following parameters:

Table 8: Analytically computed trade-offs for different protocols and different environments for soundness 2^{128} and completeness 2^{-40}, all using secp256k1. Intel/WASM are as in Table 4.

Protocol	**Intel**	**WASM**
ElGamalCommit	$\rho_{\min} = 24.4$ $\rho = 26, b = 5$	$\rho_{\min} = 19.2$ $\rho = 19, b = 7$
ElGamalCommitMultScalar	$\rho_{\min} = 20.2$ $\rho = 22, b = 6$	$\rho_{\min} = 18.0$ $\rho = 19, b = 7$

Experimental results.

Table 9: Running times for the **prover** for Fischlin in milliseconds for secp256k1; average over 1,000 executions. Intel/WASM are as in Table 4.

	$\boldsymbol{b=3}$	$\boldsymbol{b=4}$	$\boldsymbol{b=5}$	$\boldsymbol{b=6}$	$\boldsymbol{b=7}$	$\boldsymbol{b=8}$	$\boldsymbol{b=9}$
ElGamalCommit (Intel)	2.49	2.23	**2.22**	2.50	3.40	5.03	8.47
ElGamalCommit (WASM)	13.2	10.6	9.45	**9.31**	10.5	13.7	22.2
ElGamalCommitMultScalar (Intel)	5.35	4.38	**4.00**	4.47	4.87	6.10	9.65
ElGamalCommitMultScalar (WASM)	34.2	26.2	22.1	20.3	**19.6**	21.3	27.7

As we can see, the prediction was accurate for ElGamalCommit on Intel and for ElGamalCommitMultScalar on WASM, but off by one for ElGamalCommit on WASM and for ElGamalCommitMultScalar on Intel. In order to complete the picture, we also provide the running times for the verifier:

Table 10: Running times for the **verifier** for Fischlin in milliseconds for secp256k1; average over 1,000 executions. Intel/WASM are as in Table 4.

	$\boldsymbol{b = 3}$	$\boldsymbol{b = 4}$	$\boldsymbol{b = 5}$	$\boldsymbol{b = 6}$	$\boldsymbol{b = 7}$	$\boldsymbol{b = 8}$	$\boldsymbol{b = 9}$
ElGamalCommit (Intel)	1.65	1.25	1.04	2.50	0.90	0.80	0.70
ElGamalCommit (WASM)	9.23	7.06	5.67	4.97	4.8	3.8	3.58
ElGamalCommitMultScalar (Intel)	1.73	1.37	1.17	1.04	0.93	0.71	0.69
ElGamalCommitMultScalar (WASM)	9.9	7.67	6.27	5.38	4.72	4.26	3.86