Weak Fiat-Shamir Attacks on Modern Proof Systems

Quang Dao
Carnegie Mellon University

Jim Miller
Trail of Bits

Opal Wright
Trail of Bits

Paul Grubbs
University of Michigan

***Abstract*—A flurry of excitement amongst researchers and practitioners has produced modern proof systems built using novel technical ideas and seeing rapid deployment, especially in cryptocurrencies. Most of these modern proof systems use the Fiat-Shamir (F-S) transformation, a seminal method of removing interaction from a protocol with a public-coin verifier. Some prior work has shown that incorrectly applying F-S (i.e., using the so-called "weak" F-S transformation) can lead to breaks of classic protocols like Schnorr's discrete log proof; however, little is known about the risks of applying F-S incorrectly for modern proof systems seeing deployment today.**

In this paper, we fill this knowledge gap via a broad theoretical and practical study of F-S in implementations of modern proof systems. We perform a survey of open-source implementations and find 36 weak F-S implementations affecting 12 different proof systems. For four of these—Bulletproofs, Plonk, Spartan, and Wesolowski's VDF—we develop novel knowledge soundness attacks accompanied by rigorous proofs of their efficacy. We perform case studies of applications that use vulnerable implementations, and demonstrate that a weak F-S vulnerability could have led to the creation of unlimited currency in a private blockchain protocol. Finally, we discuss possible mitigations and takeaways for academics and practitioners.

I. Introduction

Proof systems—cryptographic protocols in which a prover convinces a verifier of the truth of some public statement—have seen an explosion of interest from academic researchers and practitioners. The resulting modern constructions, in particular those enjoying an additional *zero-knowledge* property, are being widely deployed in blockchain and cryptocurrency settings [5], [26], [55], [66], [68], [77], [80], [84], [94]. A critical security property shared by all proof systems is *soundness*—roughly, this guarantees a prover can only convince a verifier of the truth of actually true statements. Applications like cryptocurrencies rely on soundness to, e.g., prevent attackers from creating money out of thin air.

Most proof systems used in practice are non-interactive: they consist of a single message from the prover to the verifier. Though built using novel and varied technical tools, most modern non-interactive proof systems share a common design pattern: first, build and analyze an interactive protocol where the verifier's messages consist solely of random values (i.e., it is *public-coin*), then compile it to a non-interactive protocol using the Fiat-Shamir (F-S) transformation [35]. The transformation works by replacing the public-coin verifier with a hash function: each verifier challenge is derived by the prover by hashing the transcript of the prover's messages thus far. A standard result [76] shows that if done correctly, this transformation preserves security if the hash function is modelled as a random oracle [9].

Unfortunately, it is surprisingly easy to implement F-S incorrectly. An important subtlety, which is not often discussed, is whether it is necessary to include public information, such as the statement, in the transcript. The version of the transformation where the public information is not hashed is usually called **weak** F-S; if the public information is hashed, this is usually called **strong** F-S (or simply F-S). (See Figure 1 for an example of the differences for Schnorr's discrete log proof.) Intuitively, hashing public information ensures that the proof depends on the public information, preventing a malicious prover from *adaptively* choosing it during, or even after, generating a proof. Prior work has shown that many classic proof systems, such as Schnorr [79] and Chaum-Pedersen [25], cannot be adaptively sound if weak F-S is used; further, this lack of adaptive soundness breaks applications that use these proof systems. For example, an adaptive soundness attack on Chaum-Pedersen was shown to compromise the voting protocols Helios [13] and sVote [46].

Despite these important prior works, little is known about the risks of weak F-S for the modern proof systems being used in practice today. This gap in our knowledge is serious for at least two reasons. First, modern proof systems are built using newer and arguably more complex technical tools than classic schemes, meaning prior attacks do not easily translate. Second, since more proof systems are being deployed than ever before, the potential attack surface is much larger, and the consequences of attacks could be more severe. Thus, it is crucial to understand whether vulnerable code exists and how it could be exploited.

Our contributions. In this paper, we fill this gap with a broad study of the risks of weak F-S in modern proof systems. Our main contributions are fourfold: first, we perform an extensive survey of over 75 open-source proof system implementations that use F-S, uncovering **36** weak F-S implementations across **12** different proof systems. Second, for four of the proof systems with at least one weak F-S implementation, we construct, analyze, and implement novel knowledge soundness attacks. Third, we perform case studies of how these proof systems are used in applications, to understand whether our weak F-S attacks would have led to breaks of real systems. One case study shows that it would have been possible to create unlimited money in the Dusk Network testnet [92].

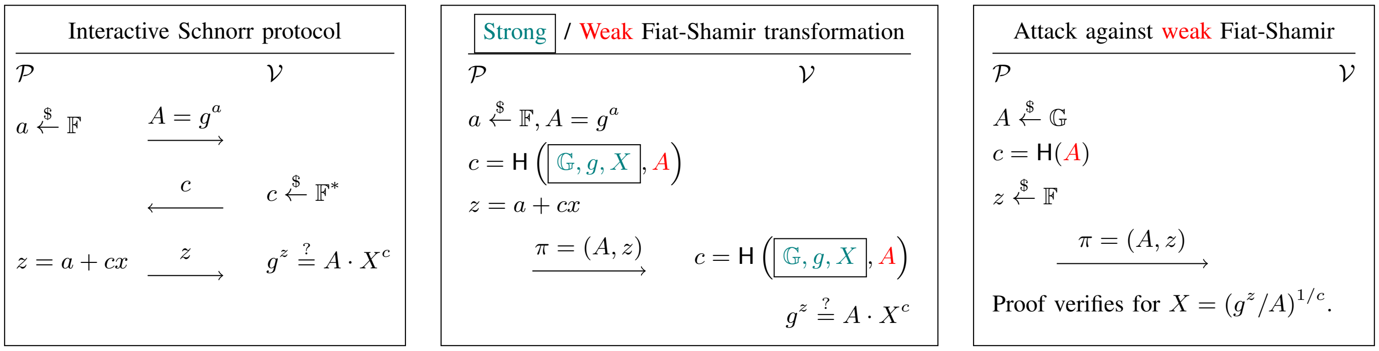
Interactive Schnorr protocol
$\mathcal{P}$
$\mathcal{V}$
$a \xleftarrow{\$} \mathbb{F}$
$A = g^a$
c
$c \xleftarrow{\$} \mathbb{F}^*$
$z = a + cx$
z
$g^z \overset{?}{=} A \cdot X^c$
Strong / Weak Fiat-Shamir transformation
$a \xleftarrow{\$} \mathbb{F}, A = g^a$
$c = \mathsf{H}\left(\boxed{\mathbb{G}, g, X}, A\right)$
$z = a + cx$
$\pi = (A, z)$
$c = \mathsf{H}\left(\boxed{\mathbb{G}, g, X}, A\right)$
$g^z \overset{?}{=} A \cdot X^c$
Attack against weak Fiat-Shamir
$A \xleftarrow{\$} \mathbb{G}$
$c = \mathsf{H}(A)$
$z \xleftarrow{\$} \mathbb{F}$
$\pi = (A, z)$
Proof verifies for $X = (g^z/A)^{1/c}$.

Fig. 1: Example weak Fiat-Shamir attack against Schnorr proofs for relation $\{((\mathbb{G}, g), X; x) \mid X = g^x\}$

Finally, we explore the landscape of mitigations, identifying design criteria and studying how proposals would apply to Merlin, a widely-used Rust library for implementing F-S.

Example of F-S for Schnorr. Before describing our contributions in more detail, we will explain the basics of applying F-S to the classic Schnorr protocol. The left-hand side of Figure 1 describes the three moves of the interactive protocol for proving knowledge of the discrete log x of an element X in a prime-order group $\mathbb{G}$ with generator g. For Schnorr, the group description and generator are examples of *public parameters* that define the set of provable statements. The group element X is the *public input* about which the prover is generating a proof, and the value x is the *witness* the prover wishes to hide.

The middle box of Figure 1 depicts the two ways of applying F-S to Schnorr. In weak F-S, only the prover's first message A is hashed. In strong F-S, the hash additionally includes the public parameters and public input. Finally, the right box of the figure depicts the *adaptive* attack (due to [13]) that is possible if weak F-S is used: by computing the public input X as a function of a randomly-generated proof, a malicious prover can convince a verifier without knowing the witness. Intuitively, strong F-S prevents this because the public input X affects the derived challenge c.

Implementation survey. Table I summarizes our implementation survey of GitHub repositories containing implementations of proof systems. We used a combination of manual search and automated dependency checking to find the repositories. Overall, we identified at least 75 repositories that attempted to implement a non-interactive proof system using F-S. Of those, 36 used weak F-S. (For space reasons, our table lists only the 54 repos of the 12 proof systems that had at least one weak F-S implementation.) After a preliminary public disclosure [65] of some of our results, many repositories were fixed and are marked as such. The main takeaway of our survey is that misuses of F-S are very widespread, and that even production-quality code written by experts—who in some cases are the creators of the proof system—implemented weak F-S. Interestingly, we found several repositories that made even more severe mistakes in implementing F-S; Section VIII contains further discussion of these cases. We followed responsible disclosure best practices in informing all repository owners about the vulnerabilities.

New attacks. For four proof systems with at least one vulnerable implementation—Bulletproofs [22], Plonk [37], Spartan [82], and Wesolowski's VDF [90]—we show that using weak F-S leads to attacks on their soundness when the prover can choose the public inputs adaptively, as a function of the proof. Importantly, our results do not invalidate the security proofs for these schemes—when given explicitly, soundness proofs for non-interactive, weak F-S variants of these protocols provide only *non-adaptive* security.

In Section IV, we show an attack on the (adaptive) knowledge soundness of the Bulletproofs aggregate range proof, which would allow crafting Pedersen commitments to values that lie outside the specific range with high probability. In Sections V and VI, we give our attacks on Plonk and Spartan, two proofs systems that prove NP-complete constraint satisfaction problems and are built using the recent *polynomial interactive oracle proofs (IOPs)* paradigm [23], [27]. Both work by having the malicious prover choose one of the public inputs to the constraint system as a function of the proof; the public input is chosen to ensure the verification equation—in both cases, a polynomial identity—holds. Finally, in Section VII we give our attack on Wesolowski's VDF. Our attack allows a malicious prover to craft a proof π for a small delay parameter t, then compute a much larger parameter $T \gg t$ for which π is valid. Thus, the prover can claim to have done T sequential squarings while having done only t. We implement our attacks for Bulletproofs, Plonk, and Wesolowski's VDF, and experimentally verify that forged proofs can be generated quickly: for example, our Bulletproofs attack can generate a forged range proof with 32-bit range in 86 milliseconds.

Aside from the attacks themselves, an important novelty of our work is that we rigorously prove all four attacks break a well-specified soundness property of the proof system. For Bulletproofs, Plonk, and Spartan, we prove our attacks violate adaptive knowledge soundness via a meta-reduction argument: roughly, we prove that if an extractor exists for our malicious prover, this extractor could be used to break a cryptographic hardness assumption like discrete log. This technique is similar to the one used in prior work [13], but applying it here

Proof System	Codebase	Weak F-S?
Bulletproofs [22]	bp-go [87]	✓
	bulletproof-js [2]	✓
	simple-bulletproof-js [83]	✓
	BulletproofSwift [20]	✓
	python-bulletproofs [78]	✓
	adjoint-bulletproofs [3]	✓
	zkSen [98]	✓
	incognito-chain [51]	✓♦
	encoins-bulletproofs [33]	✓♦
	ZenGo-X [96]	✓♦
	zkrp [52]	✓♦
	ckb-zkp [81]	✓♦
	bulletproofsrb [21]	✓♦
	monero [68]	✗
	dalek-bulletproofs [29]	✗
	secp256k1-zkp [75]	✗
	bulletproofs-ocaml [74]	✗
	tari-project [85]	✗
	Litecoin [59]	✗
	Grin [44]	✗
Bulletproofs variant [40]	dalek-bulletproofs [29]	✓♦
	cpp-lwevss [60]	✗
Sonic [61]	ebfull-sonic [18]	✓
	lx-sonic [58]	✓
	iohk-sonic [53]	✗
	adjoint-sonic [4]	✗
Schnorr [79]	noknow-python [7]	✓
Plonk [37]	anoma-plonkup [6]	✓
	gnark [17]	✓♦
	dusk-network [31]	✓♦
	snarkjs [50]	✓♦
	ZK-Garage [97]	✓♦
	plonky [67]	✗
	ckb-zkp [81]	✗
	halo2 [93]	✗
	o1-labs [71]	✗
	jellyfish [34]	✗
	matter-labs [62]	✗
	aztec-connect [8]	✗
Wesolowski's VDF [90]	0xProject [1]	✓
	Chia [69]	✓
	Harmony [47]	✓
	POA Network [70]	✓
	IOTA Ledger [54]	✓
	master-thesis-ELTE [48]	✓
Hyrax [89]	ckb-zkp [81]	✓♦
	hyraxZK [49]	✗
Spartan [82]	Spartan [64]	✓♦
	ckb-zkp [81]	✓♦
Libra [91]	ckb-zkp [81]	✓♦
Brakedown [43]	Brakedown [19]	✓
Nova [57]	Nova [63]	✓♦
Gemini [16]	arkworks-gemini [38]	✓♦
Girault [42]	zk-paillier [95]	✓♦

TABLE I: Implementations surveyed. We include every proof system with at least one vulnerable implementation, and survey all implementations for each one (except classic protocols like Schnorr and Girault). ♦ = has been fixed as of May 15, 2023.

requires new technical ideas; e.g., for Plonk and Spartan, a trivial extractor may exist for an "easy" constraint system. Our proofs for Plonk and Spartan show that knowledge soundness breaks as soon as the relation satisfies a slight strengthening of worst-case hardness; our analysis here may be of independent interest.

Application case studies. Dozen of implementations of the four proof systems we examined are vulnerable to these attacks, at least in theory. However, this does not necessarily mean the applications that use them are broken by these attacks—it could be that external application constraints prevent exploiting weak F-S. To answer this question, we next look at the applications that use vulnerable proof systems. Here our findings are more mixed. While we identified one application that is unambiguously broken by a weak F-S attack—we show in Section V-C that it would have been possible to create unlimited money in the Dusk Network testnet—our other attacks do not appear to break applications. For Spartan, we were not able to identify any vulnerable applications. For Bulletproofs, the implementations we found that were actually used in real applications were not vulnerable. Nevertheless, we give a "counterfactual" case study of the Mimblewimble protocol [55] to determine if our weak F-S attack *could* have led to an application break; we find that creating unlimited money would have been possible. For Wesolowski's VDF, constraints on the size of the delay parameter prevent our malicious proofs from breaking some applications, like the Chia blockchain, but we found at least one case (the 0x VDF verifier smart contract) where no constraints exist.

Mitigations. Finally, in Section IX we discuss how to mitigate weak F-S attacks. We explore creating tools that can detect weak F-S vulnerabilities in existing code, and also study how existing tools, such as the Merlin library for F-S [28], could be modified to make them harder to misuse. (Several vulnerable implementations we found used Merlin.) To detect weak F-S implementations, we describe how information-flow analysis could be used to ensure variables in common between the prover and verifier are hashed in the transcript. To make it harder to implement F-S incorrectly, we suggest modifying the Merlin API to force programmers to initialize protocol transcripts with public inputs, or to specify all F-S inputs and challenges upon initialization of the transcript. These approaches have some drawbacks, which we discuss in Section IX-A. We leave an implementation and evaluation of these tools to future work.

A. Related Works

This paper extends and generalizes our preliminary results, which were posted in a series of blog posts [65]. Compared to the blog posts, we perform a more comprehensive implementation survey which uncover more vulnerable implementations, give attacks for two more proof systems (Spartan and

Wesolowski's VDF), provide rigorous proofs that our attacks break security, and give new case studies of practical impacts.

Our work is about the Fiat-Shamir transformation, originally given in [35]. Our work also applies to variants of the transformation for multi-round protocols, such as the BCS transformation of [11]. We did not study protocols that use quantum variants of F-S, such as the Unruh's transformation [86]; our attacks should extend to these, but we leave the details to future work. Our attacks do not apply to proof systems that use only structured reference strings for non-interactivity, such as Groth16 [45].

Our work is indebted to the seminal paper on weak F-S by Bernhard et al. [13], which highlighted this issue and gave attacks against Schnorr and Chaum-Pedersen. A key followup to [13] that uncovered other weak F-S attacks on similar sigma protocols, that also break voting systems, is [46]. As discussed above, our work examines weak F-S in the context of proof systems built in the last decade or so, which use new and very different building blocks from older schemes: these include non-constant-round interactive protocols, such as Bulletproofs. We use a similar meta-reduction technique to [13] for analyzing our attacks, though heavily modified to account for the proof systems' ability to prove more complex (even NP-complete) relations.

Our work shows that the four non-interactive proof systems we studied are not sound in an adaptive setting. There has been some work finding different kinds of soundness bugs in proof systems; these bugs are caused by faulty proofs and apply even in the non-adaptive soundness setting. For example, [36] found a soundness bug in the BCTV SNARK construction, and [72] found a soundness bug in vnTinyRAM. Our attacks do not stem from faulty proofs, but rather from a gap between what the proofs guarantee and the security that applications require.

II. Preliminaries

A. Notation

We denote the security parameter by λ, and a negligible function in λ by $\mathsf{negl}(\lambda)$. Our relations, cryptographic objects, and adversaries all depend on λ; we often omit this dependency. We use game-based security definitions [10]; here a game $\mathsf{G}^{\mathcal{A}_1,\ldots,\mathcal{A}_n}_{\mathsf{S}_1,\ldots,\mathsf{S}_m}$ denotes a run of parties $\mathcal{A}_1,\ldots,\mathcal{A}_n$ on a pre-specified set of procedures given by $\mathsf{S}_1,\ldots,\mathsf{S}_m$, returning a bit $b \in \{0,1\}$. We denote by $\Pr\left[\mathsf{G}^{\mathcal{A}_1,\ldots,\mathcal{A}_n}_{\mathsf{S}_1,\ldots,\mathsf{S}_m}\right]$ the probability, over the random coins used by all parties and the game itself, that the game's output is 1.

We denote by $\mathbb{G}$ a group, either of prime order p or of unknown order, $\mathbb{F}$ a finite field of prime order p, and use $x \overset{\$}{\leftarrow} \mathbb{F}$ to denote uniformly sampling an element in $\mathbb{F}$. We denote vectors by boldface $\mathbf{x} \in \mathbb{F}^n$, the inner product of two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{F}^n$ by $\langle \mathbf{x}, \mathbf{y} \rangle$, the element-wise product by $\mathbf{x} \circ \mathbf{y}$, subvectors by $\mathbf{x}_{[i:j]} = (x_i, x_{i+1}, \ldots, x_j)$, vector subscripts by $\mathbf{x}_{\mathbf{a}} = (x_{a_1}, \ldots, x_{a_m})$ where $\mathbf{a} \in [n]^m$, and multi-exponentiation between $\mathbf{g} \in \mathbb{G}^n$ and $\mathbf{x} \in \mathbb{F}^n$ by $\mathbf{g}^{\mathbf{x}}$. For $y \in \mathbb{F}$, we denote $\mathbf{y}^n = (1, y, \ldots, y^{n-1})$. We write

```
Game $\mathsf{DL}^{\mathcal{A}}_{\mathbb{G}}(\lambda)$
    $h \overset{\$}{\leftarrow} \mathbb{G} \setminus \{g\}$
    $a \leftarrow \mathcal{A}(g, h)$
    return $(g^a = h)$

Game $\mathsf{DL\text{-}REL}^{\mathcal{A}}_{\mathbb{G},n}(\lambda)$
    $g_1, \ldots, g_n \overset{\$}{\leftarrow} \mathbb{G}$
    $(a_1, \ldots, a_n) \leftarrow \mathcal{A}(g_1, \ldots, g_n)$
    return $\left(\prod_{i=1}^{n} g_i^{a_i} = 1\right)$
        $\wedge\, ((a_1, \ldots, a_n) \neq 0^n)$
```

Fig. 2: Games for Discrete Log

$p(X) \in \mathbb{F}^{<d}[X]$ to denote a (univariate) polynomial of degree less than d, and $p(X) \in \mathbb{F}[\mu]$ to denote a multilinear polynomial in μ variables.

Lagrange basis. Given a finite field $\mathbb{F}$ and a subgroup $H = \langle \omega \rangle$ of order n, for every $i \in [n]$ we can define the Lagrange polynomial $\mathsf{L}_i(X)$ to be the unique polynomial of degree $n-1$ that satisfies $\mathsf{L}_i(\omega^i) = 1$ and $\mathsf{L}_i(\omega^j) = 0$ for $j \neq i$. For any vector $\mathbf{x} \in \mathbb{F}^n$, there exists a unique polynomial $\mathsf{p}(X)$ of degree at most $n-1$ that satisfies $\mathsf{p}(\omega^i) = \mathbf{x}_i$; we have the identity $\mathsf{p}(X) = \sum_{i=1}^{n} \mathsf{x}_i \cdot \mathsf{L}_i(X)$.

Multilinear extension. Given $g : \{0,1\}^\mu \to \mathbb{F}$, we define

$$\widetilde{g}(X_1, \ldots, X_\mu) \in \mathbb{F}[X_1, \ldots, X_\mu]$$

to be the unique multilinear polynomial with evaluation $\widetilde{g}(y) = g(y)$ for all $y \in \{0,1\}^\mu$, called the *multilinear extension* of g. We have the identity

$$g(X_1, \ldots, X_\mu) = \sum_{y \in \{0,1\}^\mu} g(y) \cdot \widetilde{\mathsf{eq}}(X, y),$$

where

$$\mathsf{eq}(X, Y) = \prod_{i=1}^{\mu} (X_i \cdot Y_i + (1 - X_i) \cdot (1 - Y_i)).$$

Here $\widetilde{\mathsf{eq}}(X, Y)$ is the analogue of the Lagrange basis in the multilinear setting; we have $\mathsf{eq}(x, x) = 1$ and $\mathsf{eq}(x, y) = 0$ if $x \neq y$.

B. Discrete Log Assumptions

Let $\mathbb{G}$ be a prime-order group (depending on λ), with generator g and scalar field $\mathbb{F}$.

Definition 1: We say that the discrete log (DL) *assumption holds for* $\mathbb{G}$ *if for all PPT adversaries* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\mathbf{Adv}^{\mathsf{DL}}_{\mathbb{G}}(\mathcal{A}) := \Pr\left[\mathsf{DL}^{\mathcal{A}}_{\mathbb{G}}(\lambda)\right].$$

We say that the discrete log relation (DL-REL) *assumption holds for* $\mathbb{G}$ *if for all PPT adversaries* $\mathcal{A}$ *and all* $n \in \mathbb{N}$*, the following probability is negligible in* λ*:*

$$\mathbf{Adv}^{\mathsf{DL\text{-}REL}}_{\mathbb{G},n}(\mathcal{A}) := \Pr\left[\mathsf{DL\text{-}REL}^{\mathcal{A}}_{\mathbb{G},n}(\lambda)\right].$$

The two discrete log assumptions are tightly related [41].

Lemma 1: For every PPT adversary $\mathcal{A}$ against DL-REL, there exists a PPT adversary $\mathcal{B}$ against DL, nearly as efficient as $\mathcal{A}$, such that

$$\mathbf{Adv}^{\mathsf{DL\text{-}REL}}_{\mathbb{G},n}(\mathcal{A}) \le \mathbf{Adv}^{\mathsf{DL}}_{\mathbb{G}}(\mathcal{B}) + \frac{1}{|\mathbb{F}|}.$$

C. Interactive Arguments

An *interactive argument* for an NP relation $\mathcal{R}$ is a tuple of PPT algorithms $\Pi = (\mathsf{Setup}, \mathcal{P}, \mathcal{V})$. Here $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$ produces public parameters pp given a security parameter 1^λ, and $\langle \mathcal{P}(w), \mathcal{V}\rangle(\mathsf{pp}, x) \to \{0,1\}$ is an interactive protocol whereby the prover $\mathcal{P}$, holding a witness w, interacts with the verifier $\mathcal{V}$ on common input (pp, x) to convince $\mathcal{V}$ that $(x, w) \in \mathcal{R}$. At the end, $\mathcal{V}$ outputs a bit to accept or reject the proof.

We require that interactive arguments satisfy *completeness* and *knowledge soundness*. Completeness states that for every $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ and $(x, w) \in \mathcal{R}$, we have $\langle \mathcal{P}(w), \mathcal{V}\rangle(\mathsf{pp}, x) \to 1$. Knowledge soundness states that there exists an expected polynomial time extractor $\mathcal{E}$ such that for any stateful PPT adversary $\mathcal{P}^*$, the probability that $\mathcal{P}^*$ manages to convince $\mathcal{V}$ on an input x chosen by $\mathcal{P}^*$, yet $\mathcal{E}$ cannot find a witness w for x, is negligible. Here $\mathcal{E}$ gets black-box access to each of the next-message functions of $\mathcal{P}^*$ in the interactive protocol.

The interactive arguments we consider are *public-coin*, meaning that in each round the verifier $\mathcal{V}$ samples its message uniformly at random from some challenge space. Such protocols have a $(r+1)$-round format where $\mathcal{P}$ sends the first and last messages. In particular, the transcript is of the form $(a_1, c_1, \ldots, a_r, c_r, a_{r+1})$, where $(a_1, \ldots, a_{r+1})$ are messages sent by $\mathcal{P}$ and $(c_1, \ldots, c_r)$ are challenges sent by $\mathcal{V}$.

D. Non-Interactive Arguments in the ROM

The *Fiat-Shamir transformation* (see Section II-E) is often used to compile public-coin interactive arguments into their *non-interactive* versions in the *random oracle model* (ROM) [9]. We denote the random oracle by $\mathsf{H} : \{0,1\}^* \to \{0,1\}^*$.

Definition 2: *A* non-interactive argument of knowledge *(NARK) in the ROM for a NP relation $\mathcal{R}$ is a tuple of PPT algorithms $\Pi = (\mathsf{Setup}, \mathcal{P}, \mathcal{V})$, with $\mathcal{P}, \mathcal{V}$ having black-box access to a random oracle H, with the following syntax:*

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$ *: generates the public parameters for $\mathcal{R}$,*
- $\mathcal{P}^{\mathsf{H}}(\mathsf{pp}, x, w) \to \pi$ *: generates a proof,*
- $\mathcal{V}^{\mathsf{H}}(\mathsf{pp}, x, \pi) \to \{0,1\}$ *: checks whether a proof π is valid with respect to* pp *and input x.*

We require NARKs to satisfy the following properties.

- *Completeness.* For every $(x, w) \in \mathcal{R}$,
$$\Pr\left[\mathcal{V}^{\mathsf{H}}(\mathsf{pp}, x, \pi) = 1 \ : \ \begin{matrix} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ \pi \leftarrow \mathcal{P}^{\mathsf{H}}(\mathsf{pp}, x, w) \end{matrix}\right] = 1.$$
- *Knowledge soundness.* For every PPT adversary $\mathcal{P}^*$, there exists an extractor $\mathcal{E}$ running in expected polynomial time such that the following probability is $\mathsf{negl}(\lambda)$:
$$\mathbf{Adv}^{\mathsf{KS}}_{\Pi,\mathcal{R}}(\mathcal{E}, \mathcal{P}^*) := \left|\Pr[\mathsf{KS}^{\mathcal{P}^*}_{0,\Pi}(\lambda)] - \Pr[\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\Pi,\mathcal{R}}(\lambda)]\right|.$$
The KS games are defined in Figure 3.

```
Game $\mathsf{KS}^{\mathcal{P}^*}_{0,\Pi}(\lambda)$
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(x, \pi) \leftarrow (\mathcal{P}^*)^{\mathsf{H}}(\mathsf{pp})$
$b \leftarrow \mathcal{V}^{\mathsf{H}}(\mathsf{pp}, x, \pi)$
return $b$
```

```
Game $\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\Pi,\mathcal{R}}(\lambda)$
$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$(x, \pi) \leftarrow (\mathcal{P}^*)^{\mathsf{H}}(\mathsf{pp})$
$b \leftarrow \mathcal{V}^{\mathsf{H}}(\mathsf{pp}, x, \pi)$
$w \leftarrow \mathcal{E}(\mathcal{P}^*, \mathsf{pp}, x, \pi)$
return $b \wedge (x, w) \in \mathcal{R}$
```

Fig. 3: Knowledge soundness security games. Here the extractor $\mathcal{E}$ is given the description of $\mathcal{P}^*$; in particular, it may rewind $\mathcal{P}^*$ and reprogram the random oracle H.

We note that our knowledge soundness definition is both *non-black-box*, where the extractor may depend on (the code of) the malicious prover $\mathcal{P}^*$, and *adaptive*, meaning the malicious prover $\mathcal{P}^*$ can choose the pair (x, π) at the same time. The adaptive strengthening is often necessary in practice, as evidenced by our case studies (e.g., see Section IV-C). We also discuss the situation where $\mathcal{P}^*$ may also influence the public parameters pp in Section VIII. On the other hand, non-black-box extraction is a weaker extractability requirement [24][1], including extracting using non-falsifiable knowledge assumptions [12], [39], [45], [73]. Looking ahead, our results for Plonk and Spartan ruling out non-black-box extraction for "sufficiently hard" relations will also rule out black-box extraction.

E. The Fiat-Shamir Transformation

We define both variants (weak and strong) of the *Fiat-Shamir transformation.*

Definition 3: *Let $\Pi = (\mathsf{Setup}, \mathcal{P}, \mathcal{V})$ be a public-coin interactive argument with transcript of the form $\mathsf{tr} = (a_1, c_1, \ldots, a_r, c_r, a_{r+1})$. The* strong Fiat-Shamir transformation *turns Π into a non-interactive argument Π_{sFS} where:*

- $\mathsf{Setup}_{\mathsf{sFS}}(1^\lambda)$ *is the same as* $\mathsf{Setup}(1^\lambda)$*,*
- *the prover $\mathcal{P}_{\mathsf{sFS}}$, on input (pp, x, w), invokes $\mathcal{P}(\mathsf{pp}, x, w)$, and instead of asking the verifier for challenge c_i in round i, queries the random oracle to get*
$$c_i \leftarrow \mathsf{H}(\mathsf{pp}, x, a_1, \ldots, a_i) \quad \forall\, i = 1, \ldots, r.$$
$\mathcal{P}_{\mathsf{sFS}}$ *then outputs the proof $\pi = (a_1, \ldots, a_r, a_{r+1})$.*
- *the verifier $\mathcal{V}_{\mathsf{sFS}}$, on input (pp, x, π), derives challenges c_i by querying the random oracle as above, then runs $\mathcal{V}(\mathsf{pp}, x)$ on transcript $(a_1, c_1, \ldots, a_r, c_r, a_{r+1})$ and outputs what $\mathcal{V}$ outputs.*

The weak Fiat-Shamir transformation *is similar, except that we omit the public parameters* pp *and the input x from the hash, so that*
$$c_i = \mathsf{H}(a_1, \ldots, a_i) \quad \forall\, i = 1, \ldots, r.$$
We denote the weak Fiat-Shamir transformed argument by $\Pi_{\mathsf{wFS}} = (\mathsf{Setup}, \mathcal{P}_{\mathsf{wFS}}, \mathcal{V}_{\mathsf{wFS}})$.

[1]In contrast, *black-box* extraction requires a single extractor that works for all malicious provers, and may only rely on its input-output behavior.

F. *Polynomial Interactive Oracle Proofs*

We describe the formalism of *polynomial IOPs* [23], [27] that underlies Plonk and Spartan.

Polynomial IOP. A *(public-coin) polynomial IOP* for a NP relation $\mathcal{R}$ (depending on a field $\mathbb{F}$) is a tuple of PPT algorithms $(\mathcal{I}, \mathcal{P}, \mathcal{V})$ with the following protocol format. In the preprocessing phase, the indexer $\mathcal{I}(\mathbb{F}, \mathcal{R})$ outputs a list of preprocessed *polynomial oracles* $\mathbb{i}$. In the interaction phase, the prover $\mathcal{P}$ is given $(\mathbb{i}, \mathbb{x}, \mathbb{w})$ and the verifier $\mathcal{V}$ is given $(\mathbb{i}, \mathbb{x})$. In each round i, $\mathcal{P}$ sends a list of polynomial oracles $\mathbb{p}_i$, and $\mathcal{V}$ responds with a random challenge c_i. In the query phase, $\mathcal{V}$ may query any of the polynomial oracle p, obtained as part of $\mathbb{i}$ or $\mathbb{p}_i$ for some round i, at any evaluation point z to get the corresponding evaluation $\mathsf{p}(z)$. $\mathcal{V}$ then outputs accept or reject. *Completeness* and *knowledge soundness* for polynomial IOPs are defined similarly to interactive arguments; see [27] for full definitions.

Polynomial Commitment Scheme. A *polynomial commitment scheme* (PC) is a tuple of PPT algorithms $\mathsf{PC} = (\mathsf{Setup}, \mathsf{Commit})$ and an interactive argument Open with the following syntax:

- $\mathsf{Setup}(1^\lambda, \mu, D) \to \mathsf{pp}$: sets up public parameters pp given number of variables μ and maximum individual degree D,
- $\mathsf{Commit}(\mathsf{pp}, \mathsf{p}; \omega) \to [\mathsf{p}]$: outputs a commitment $[\mathsf{p}]$ to a polynomial $\mathsf{p} \in \mathbb{F}^{\leq D}[X_1, \ldots, X_\mu]$, using randomness ω,
- $\mathsf{Open} := \langle \mathcal{P}_{\mathsf{PC}}(\mathsf{p}, \omega), \mathcal{V}_{\mathsf{PC}} \rangle(\mathsf{pp}, [\mathsf{p}], x, v) \to \{0, 1\}$ is a public-coin interactive argument for the relation $\mathsf{p}(x) = v$ and $[\mathsf{p}] = \mathsf{Commit}(\mathsf{pp}, \mathsf{p}; \omega)$.

We consider two types of PCs in our paper, one for univariate polynomials ($\mu = 1$) and one for multilinear polynomials ($D = 1$). We define *completeness* and *knowledge soundness* of PC to be the corresponding property for Open.

Compiling to non-interactive arguments. Any polynomial IOP can be composed with any polynomial commitment scheme PC to form a public-coin interactive argument $\Pi = (\mathsf{Setup}, \mathcal{P}, \mathcal{V})$, which can then be turned non-interactive via Fiat-Shamir. The former step is done as follows:

- $\mathsf{Setup}(1^\lambda)$: runs $\mathsf{PC.Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{PC}}$, $\mathcal{I}(\mathbb{F}, \mathcal{R}) \to \mathbb{i}$, and $\mathsf{PC.Commit}(\mathsf{pp}_{\mathsf{PC}}, \mathsf{i}) \to [\mathsf{i}]$ for all $\mathsf{i} \in \mathbb{i}$. Outputs $\mathsf{pp} = (\mathsf{pp}_{\mathsf{PC}}, ([\mathsf{i}])_{\mathsf{i} \in \mathbb{i}})$.
- $\langle \mathcal{P}(\mathbb{w}), \mathcal{V} \rangle(\mathsf{pp}, \mathbb{x})$: emulate the interaction phase of the polynomial IOP, with $\mathcal{P}$ sending a polynomial commitment $[\mathsf{p}]$ instead of an oracle for each polynomial p. $\mathcal{P}, \mathcal{V}$ then emulate the query phase, with each query $v \leftarrow \mathsf{p}(z)$ replaced by $\mathcal{P}$ sending the evaluation v, followed by an execution of $\mathsf{PC.Open}$ to prove that $v = \mathsf{p}(z)$.

III. Attack Overview

In this section, we give a common template for our attacks against weak Fiat-Shamir transformations, with the attack on Schnorr (see Figure 1) as an explicit example. In the following sections, we will use this template to instantiate attacks against Bulletproofs, Plonk, Spartan, and Wesolowski's VDF. Since the details vary greatly between each proof system, we urge the reader to cross-reference the template here with the details of each attack.

1) First, we identify the part of the public statement that is *not* included in the Fiat-Shamir transformation (e.g., certain public parameters or public inputs to a circuit). For Schnorr, this includes the public input X.
2) We then identify the verification step that relies on these public values. For Schnorr, the check is $g^z \overset{?}{=} A \cdot X^c$.
3) We select arbitrary witness values and randomness for proof generation, then use them to compute all intermediate proof values. For Schnorr, we sample random $A \overset{\$}{\leftarrow} \mathbb{G}$ and $z \overset{\$}{\leftarrow} \mathbb{F}$.
4) Finally, we use the intermediate values from step 3 to solve for the public value that will always pass the verification step from step 2. For Schnorr, we set $X = (g^z/A)^{1/c}$.

We leave to future work the task of using this template to instantate attacks against other proof systems, especially the ones appearing in Table I for which we did not give attacks in this paper.

IV. Bulletproofs

In this section, we describe an attack that is possible when the Bulletproofs aggregate range proof protocol (BP-ARP) [22] is instantiated with weak Fiat-Shamir and consider the practical impacts of such an attack on MimbleWimble [55].

A. *Protocol Description*

Aggregate range proof relation. In an aggregate range proof, the public input is a vector of commitments $\mathbf{V} = (V_i)_{i \in [m]}$, and the prover's task is to show that V_i is a commitment of a value v_i belonging to small range $[0, 2^n - 1]$. Formally, we consider the relation

$$\mathcal{R}_{\mathsf{BP\text{-}ARP}} = \left\{ \begin{matrix} ((m, n, \mathbf{g}, \mathbf{h}, g, h, u), \mathbf{V}, (\mathbf{v}, \boldsymbol{\gamma})) : \\ V_j = g^{v_j} h^{\gamma_j} \ \wedge \ v_j \in [0, 2^n - 1] \ \forall j \in [1, m] \end{matrix} \right\}$$

Here m, n are powers of 2, and $\mathbf{g}, \mathbf{h} \in \mathbb{G}^{m \cdot n}$, $g, h, u \in \mathbb{G}$ are generators with unknown discrete log relations.

Converting to inner product argument. To prove $v_i \in [0, 2^n - 1]$ for all $i \in [m]$, the prover will commit to the bit decomposition $\mathbf{a}_L$ of $v_1, \ldots, v_m$ and prove that: (1) $\mathbf{a}_L \circ \mathbf{a}_R = 0$ where $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^{m \cdot n}$, and (2) $\langle (\mathbf{a}_L)_{[(i-1)n, in-1]}, \mathbf{2}^n \rangle = v_i$ for all $i \in [m]$. To achieve zero-knowledge, the prover also samples blinding vectors $\mathbf{s}_L, \mathbf{s}_R \overset{\$}{\leftarrow} \mathbb{F}^{m \cdot n}$ and computes two vector polynomials $\ell(X), r(X) \in \mathbb{F}^{m \cdot n}[X]$, which encodes all checks above into a single inner product claim. Finally, the inner product claim is proved using the Bulletproofs' inner product argument BP-IPA.

We describe the protocol BP-ARP in Figure 5, which uses the BP-IPA subprotocol in Figure 4. The single range proof protocol BP-RP is a special case of BP-ARP when $m = 1$.

Inner Product Relation. Given a power of two $n = 2^k$ and vectors of group elements $\mathbf{g}, \mathbf{h} \in \mathbb{G}^n$,

$$\mathcal{R}_{\mathsf{IPA}} = \left\{ ((n, \mathbf{g}, \mathbf{h}, u), P, (\mathbf{a}, \mathbf{b})) \mid P = \mathbf{g}^{\mathbf{a}} \mathbf{h}^{\mathbf{b}} u^{\langle \mathbf{a}, \mathbf{b} \rangle} \right\}.$$

Interaction Phase. Set $n_0 \leftarrow n, \mathbf{g}^{(0)} \leftarrow \mathbf{g}, \mathbf{h}^{(0)} \leftarrow \mathbf{h}, P^{(0)} \leftarrow P, \mathbf{a}^{(0)} \leftarrow \mathbf{a}, \mathbf{b}^{(0)} \leftarrow \mathbf{b}$.

For $i = 1, \dots, k$:

1) $\mathcal{P}$ computes $n_i \leftarrow n_{i-1}/2$, $c_L \leftarrow \langle \mathbf{a}^{(i)}_{[:n_i]}, \mathbf{b}^{(i)}_{[n_i:]} \rangle$, $c_R \leftarrow \langle \mathbf{a}^{(i)}_{[n_i:]}, \mathbf{b}^{(i)}_{[:n_i]} \rangle$, and

$$L_i \leftarrow \left(\mathbf{g}^{(i-1)}_{[n_i:]} \right)^{\mathbf{a}^{(i)}_{[:n_i]}} \left(\mathbf{h}^{(i-1)}_{[:n_i]} \right)^{\mathbf{b}^{(i)}_{[n_i:]}} u^{c_L},$$

$$R_i \leftarrow \left(\mathbf{g}^{(i-1)}_{[:n_i]} \right)^{\mathbf{a}^{(i)}_{[n_i:]}} \left(\mathbf{h}^{(i-1)}_{[n_i:]} \right)^{\mathbf{b}^{(i)}_{[:n_i]}} u^{c_R}.$$

$\mathcal{P}$ sends L_i, R_i to $\mathcal{V}$.

2) $\mathcal{V}$ sends challenge $x_i \xleftarrow{\$} \mathbb{F}^*$.

3) $\mathcal{P}$, $\mathcal{V}$ both compute

$$\mathbf{g}^{(i)} \leftarrow \left(\mathbf{g}^{(i-1)}_{[:n_i]} \right)^{x_i^{-1}} \circ \left(\mathbf{g}^{(i-1)}_{[n_i:]} \right)^{x_i},$$

$$\mathbf{h}^{(i)} \leftarrow \left(\mathbf{h}^{(i-1)}_{[:n_i]} \right)^{x_i} \circ \left(\mathbf{h}^{(i-1)}_{[n_i:]} \right)^{x_i^{-1}},$$

$$P^{(i)} \leftarrow L_i^{x_i^2} P^{(i-1)} R_i^{x_i^{-2}}.$$

4) $\mathcal{P}$ computes $\mathbf{a}^{(i)} \leftarrow \mathbf{a}^{(i-1)}_{[:n_i]} \cdot x_i^{-1} + \mathbf{a}^{(i-1)}_{[n_i:]} \cdot x_i$ and $\mathbf{b}^{(i)} \leftarrow \mathbf{b}^{(i-1)}_{[:n_i]} \cdot x_i + \mathbf{b}^{(i-1)}_{[n_i:]} \cdot x_i^{-1}$.

After k rounds, $\mathcal{P}$ sends $\mathbf{a}^{(k)}, \mathbf{b}^{(k)}$ to $\mathcal{V}$.

Verification. $\mathcal{V}$ checks whether

$$P^{(k)} \overset{?}{=} \left(\mathbf{g}^{(k)} \right)^{\mathbf{a}^{(k)}} \left(\mathbf{h}^{(k)} \right)^{\mathbf{b}^{(k)}} u^{\mathbf{a}^{(k)} \cdot \mathbf{b}^{(k)}}.$$

Fig. 4: Bulletproofs' Inner Product Argument BP-IPA

B. Attack Explanation

When BP-ARP is instantiated with a weak Fiat-Shamir transformation, the challenges are derived without hashing the commitments $\mathbf{V}$. In this case, we describe an attack against BP-ARP$_{\mathsf{wFS}}$ in Figure 6. Our attack differs from an honest prover's algorithm in two ways—first, we sample t_1, t_2, τ_x uniformly at random, and second, we choose v_i, γ_i for $i \in [m]$ after computing the proof π. Our attack extends to the single (i.e. non-aggregate) range proof as well.

Correctness and performance. We show that our attack produces accepting proofs. Recall from Figure 5 that the verifier for BP-ARP$_{\mathsf{wFS}}$ checks the following: (1) whether $\pi_{\mathsf{BP\text{-}IPA}}$ is accepting, and (2) whether

$$g^{\hat{t}} h^{\tau_x} = \mathbf{V}^{(z^2, \dots, z^{m+1})} \cdot g^{\delta(y,z)} \cdot T_1^x \cdot T_2^{x^2} \tag{2}$$

Since our attack uses a valid witness $(\mathbf{l}, \mathbf{r})$ to generate $\pi_{\mathsf{BP\text{-}IPA}}$, this proof will be accepted by the verifier. Our choice of v_i, γ_i for $i \in [m]$ in step 8 of our attack then ensures that Equation 2 holds as well.

We implemented our attack in about 100 lines of Go, and verified that our forged proofs are accepted by zkrp [52].

Public Parameters. $(m, n, \mathbf{g}, \mathbf{h}, g, h, u)$.

Public Input. $(V_i)_{i \in [m]}$ **Witness.** $(v_i, \gamma_i)_{i \in [m]}$.

Interaction Phase.

1) $\mathcal{P}$ samples $\alpha, \rho \xleftarrow{\$} \mathbb{F}$, $\mathbf{s}_L, \mathbf{s}_R \xleftarrow{\$} \mathbb{F}^{m \cdot n}$ and computes $\mathbf{a}_L \in \{0,1\}^{m \cdot n}$ such that

$$\langle (\mathbf{a}_L)_{[(j-1)n, jn-1]}, \mathbf{2}^n \rangle = v_j \ \forall j \in [1, m],$$

$$\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^{m \cdot n},$$

$$A = h^{\alpha} \mathbf{g}^{\mathbf{a}_L} \mathbf{h}^{\mathbf{a}_R}, \qquad S = h^{\rho} \mathbf{g}^{\mathbf{s}_L} \mathbf{h}^{\mathbf{s}_R}.$$

$\mathcal{P}$ sends A, S to $\mathcal{V}$.

2) $\mathcal{V}$ sends challenges $y, z \xleftarrow{\$} \mathbb{F}^*$.

3) $\mathcal{P}$ samples $\tau_1, \tau_2 \xleftarrow{\$} \mathbb{F}$ and computes

$$\ell(X) = (\mathbf{a}_L - z \cdot \mathbf{1}^{m \cdot n}) + \mathbf{s}_L \cdot X,$$

$$r(X) = \mathbf{y}^{m \cdot n} \circ (\mathbf{a}_R + z \cdot \mathbf{1}^{m \cdot n} + \mathbf{s}_R \cdot X) + \sum_{j=1}^{m} z^{j+1} \cdot \left(\mathbf{0}^{(j-1)n} \| \mathbf{2}^n \| \mathbf{0}^{(m-j)n} \right),$$

$$t(X) = \langle \ell(X), r(X) \rangle = t_0 + t_1 \cdot X + t_2 \cdot X^2,$$

$$T_1 = g^{t_1} h^{\tau_1}, \qquad T_2 = g^{t_2} h^{\tau_2}.$$

$\mathcal{P}$ sends T_1, T_2 to $\mathcal{V}$.

4) $\mathcal{V}$ sends challenge $x \xleftarrow{\$} \mathbb{F}^*$.

5) $\mathcal{P}$ computes

$$\mathbf{l} = \ell(x), \quad \mathbf{r} = r(x), \quad \hat{t} = \langle \mathbf{l}, \mathbf{r} \rangle, \quad \mu = \alpha + \rho \cdot x,$$

$$\tau_x = \tau_2 \cdot x^2 + \tau_1 \cdot x + \sum_{j=1}^{m} z^{j+1} \cdot \gamma_j.$$

$\mathcal{P}$ sends $\hat{t}, \tau_x, \mu$ to $\mathcal{V}$.

6) $\mathcal{V}$ sends challenge $w \xleftarrow{\$} \mathbb{F}^*$.

7) $\mathcal{P}, \mathcal{V}$ both compute $\mathbf{h}' = \mathbf{h}^{\mathbf{y}^{-m \cdot n}}$, $u' = u^w$, and

$$P' = h^{-\mu} \cdot A \cdot S^x \cdot \mathbf{g}^{-z \cdot \mathbf{1}^{m \cdot n}} \cdot (\mathbf{h}')^{z \cdot \mathbf{y}^{m \cdot n}} \cdot \prod_{j=1}^{m} (\mathbf{h}')^{z^{j+1} \cdot \mathbf{2}^n}_{[(j-1)n, jn-1]} (u')^{\hat{t}}.$$

8) $\mathcal{P}, \mathcal{V}$ engage in BP-IPA for the triple $((m \cdot n, \mathbf{g}, \mathbf{h}', u'), P', (\mathbf{l}, \mathbf{r}))$.

Verification.

1) $\mathcal{V}$ rejects if BP-IPA fails.

2) $\mathcal{V}$ computes

$$\delta(y, z) = (z - z^2) \cdot \langle \mathbf{1}^{m \cdot n}, \mathbf{y}^{m \cdot n} \rangle - \sum_{j=1}^{m} z^{j+2} \cdot \langle \mathbf{1}^n, \mathbf{2}^n \rangle,$$

$$R = \mathbf{V}^{z^2 \cdot \mathbf{z}^m} \cdot g^{\delta(y,z)} \cdot T_1^x \cdot T_2^{x^2}.$$

3) $\mathcal{V}$ checks whether $g^{\hat{t}} h^{\tau_x} \overset{?}{=} R$.

Fig. 5: Bulletproofs' Aggregate Range Proof BP-ARP

We benchmarked forged proof generation on an Intel Core i9 running at 2.4 GHz with 16 GB of RAM. Our implementation was able to generate single range proofs (i.e. $m = 1$) for 8-bit ranges in about 23.9 milliseconds, for 16-bit ranges in 44.7 milliseconds, and for 32-bit ranges in 86.0 milliseconds. Due to limitations of the zkrp library, larger ranges could not be tested.

```
0) Initialize empty proof $\pi = \epsilon$.
1) Sample $\mathbf{a}_L \xleftarrow{\$} \{0,1\}^n$, $\alpha, \rho \xleftarrow{\$} \mathbb{F}$, $\mathbf{s}_L, \mathbf{s}_R \xleftarrow{\$} \mathbb{F}^n$, and compute
      $\mathbf{a}_R = \mathbf{a}_L - \mathbf{1}^n, \quad A \leftarrow h^\alpha \mathbf{g}^{\mathbf{a}_L} \mathbf{h}^{\mathbf{a}_R}, \quad S \leftarrow h^\rho \mathbf{g}^{\mathbf{s}_L} \mathbf{h}^{\mathbf{s}_R}.$
   Append $A, S$ to $\pi$.
2) Query challenges $y, z \leftarrow \mathsf{H}(\pi)$.
3) Sample $t_1, t_2 \xleftarrow{\$} \mathbb{F}$, $\tau_1, \tau_2 \xleftarrow{\$} \mathbb{F}$ and compute
      $T_1 \leftarrow g^{t_1} h^{\tau_1}, \qquad T_2 \leftarrow g^{t_2} h^{\tau_2}.$
   Append $T_1, T_2$ to $\pi$.
4) Query challenge $x \leftarrow \mathsf{H}(\pi)$.
5) Compute
      $\mathbf{l} \leftarrow (\mathbf{a}_L - z \cdot \mathbf{1}^n) + \mathbf{s}_L \cdot x,$
      $\mathbf{r} \leftarrow \mathbf{y}^n \circ (\mathbf{a}_R + z \cdot \mathbf{1}^n + \mathbf{s}_R \cdot x) + z^2 \cdot \mathbf{2}^n,$
      $\hat{t} = \langle \mathbf{l}, \mathbf{r} \rangle, \qquad \mu \leftarrow \alpha + \rho \cdot x.$
   Sample $\tau_x \xleftarrow{\$} \mathbb{F}$ and append $\hat{t}, \tau_x, \mu$ to $\pi$.
6) Query challenge $w \leftarrow \mathsf{H}(\pi)$.
7) Compute $\mathbf{h}'$, $u'$, $P'$ as in Figure 5, and a proof $\pi_{\mathsf{BP\text{-}IPA}}$ for
   the statement $P' = \mathbf{g}^{\mathbf{l}} (\mathbf{h}')^{\mathbf{r}} (u')^{\hat{t}}$. Append $\pi_{\mathsf{BP\text{-}IPA}}$ to $\pi$.
8) Choose $v_1, \dots, v_m, \gamma_1, \dots, \gamma_m \in \mathbb{F}$ such that
      $v_1 z^2 + \cdots + v_m z^{m+1} = \hat{t} - t_1 \cdot x - t_2 \cdot x^2 - \delta(y,z),$   (1)
      $\gamma_1 z^2 + \cdots + \gamma_m z^{m+1} = \tau_x - \tau_2 \cdot x^2 - \tau_1 \cdot x.$
   Set $V_i = g^{v_i} h^{\gamma_i}$ for all $i \in [m]$ and $\mathbf{V} = (V_i)_{i \in [m]}$.
9) Output $(\mathbf{V}, \pi)$.
```

Fig. 6: Weak Fiat-Shamir Attack Against $\mathsf{BP\text{-}ARP_{wFS}}$

Provable insecurity. We show that $\mathsf{BP\text{-}ARP_{wFS}}$ is not knowledge sound if the discrete log relation assumption holds in the underlying group, and if $2^n/|\mathbb{F}|$ is negligible. (Note that this is usually the case in practice, with typical parameters of $n \le 64$ and $|\mathbb{F}| \ge 2^{256}$.) The intuition is that at least one of the values v_i computed by the malicious prover falls outside the range $[0, 2^n - 1]$ with overwhelming probability. Hence no efficient extractor could recover values in the range consistent with the commitments, since that would lead to a non-trivial discrete log relation.

Theorem 4: *Assume* $\mathbb{G}$ *satisfies* $\mathsf{DL\text{-}REL}$, *and that* $2^n/|\mathbb{F}| = \mathsf{negl}(\lambda)$. *Then* $\mathsf{BP\text{-}ARP_{wFS}}$ *is not knowledge sound.*

Proof: Denote by $\mathcal{P}^*$ the weak Fiat-Shamir malicious prover described in Figure 6, with the following specification for step 8: $\mathcal{P}^*$ chooses $v_2, \dots, v_m$ uniformly at random, then sets v_1 to satisfy Equation 1. Since $\mathcal{P}^*$ always outputs accepting proofs, we have $\Pr\left[\mathsf{KS}^{\mathcal{P}^*}_{0,\mathsf{BP\text{-}ARP_{wFS}}}\right] = 1$. We will show that for every extractor $\mathcal{E}$, there exists an adversary $\mathcal{A}$, nearly as efficient as $\mathcal{E}$, against $\mathsf{DL\text{-}REL}$ such that

$$\Pr\left[\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\mathsf{BP\text{-}ARP_{wFS}},\mathcal{R}}\right] \le \mathbf{Adv}^{\mathsf{DL\text{-}REL}}_{\mathbb{G},2}(\mathcal{A}) + \mathsf{negl}(\lambda).$$

Thus, if $\mathsf{DL\text{-}REL}$ holds in $\mathbb{G}$, then $\mathcal{E}$ has a negligible chance of outputting a valid witness, and thus $\mathsf{BP\text{-}ARP_{wFS}}$ cannot be knowledge sound against $\mathcal{P}^*$. Before we describe $\mathcal{A}$, we note the following fact about the distribution of v_1. By construction, $\mathcal{P}^*$ chooses v_1 to be the unique value such that

$$v_1 = z^{-2} \cdot \left(\hat{t} - t_1 \cdot x - t_2 \cdot x^2 - \delta(y,z)\right) - v_2 z - \cdots - v_m z^{m-1}.$$

Since $t_1, t_2, v_2, \dots, v_m$ are sampled uniformly at random, it follows that v_1 is uniformly distributed. Since $2^n/|\mathbb{F}| = \mathsf{negl}(\lambda)$, we have $v_1 \in [0, 2^n - 1]$ with negligible probability. The adversary $\mathcal{A}$ now works as follows: first, it receives generators $g, h \xleftarrow{\$} \mathbb{G}$ in the $\mathsf{DL\text{-}REL}$ game. $\mathcal{A}$ then samples extra random generators $\mathbf{g}, \mathbf{h}, u$ and sets up the game $\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\mathsf{BP\text{-}ARP_{wFS}},\mathcal{R}}$ with $\mathsf{pp} = (m, n, \mathbf{g}, \mathbf{h}, g, h, u)$. $\mathcal{A}$ runs $\mathcal{P}^*$ once to produce $(\mathbf{V}, \pi)$, then gives $\mathcal{E}$ the description of $\mathcal{P}^*$ along with $(\mathbf{V}, \pi)$. When $\mathcal{E}$ returns a witness $(v'_i, \gamma'_i)_{i \in [m]}$, $\mathcal{A}$ returns $(v_1 - v'_1, \gamma_1 - \gamma'_1)$ in the $\mathsf{DL\text{-}REL}$ game. If $\mathcal{E}$ outputs a valid witness, we have a discrete log relation $g^{v_1} h^{\gamma_1} = V_1 = g^{v'_1} h^{\gamma'_1}$ with $v'_1 \in [0, 2^n - 1]$. As mentioned above, we know that $v_1 \in [0, 2^n - 1]$ with negligible probability; as long as that does not happen, $\mathcal{A}$ wins whenever $\mathcal{E}$ outputs a valid witness. This concludes our proof.

C. Practical Impacts

We surveyed 20 implementations of Bulletproofs and 2 implementations of a Bulletproofs variant [40] to determine if they were vulnerable to a weak Fiat-Shamir attack. Of the 22 codebases surveyed, we found 14 of them to be vulnerable. Of these 14 vulnerable implementations, 7 of them appear to be more experimental implementations, describing themselves as "university projects" or "proofs of concept." 5 of the vulnerable implementations, which have now been fixed, were developed by organizations, seemingly with the intent of being used. We believe it is likely that this high fraction of vulnerable implementations is the result of a typo (which has been fixed) in the original Bulletproofs paper, which specified a weak Fiat-Shamir implementation. Most of the 7 non-vulnerable implementations, on the other hand, were audited and maintained by organizations with the intent of using them in production.

Attacking applications that use weak Fiat-Shamir. To understand how our attack on the soundness of Bulletproofs could lead to attacks on applications that use vulnerable implementations, we surveyed the applications that use the Bulletproofs implementations in our repositories. The two main applications represented are both privacy-preserving payments protocols: Monero [68] and MimbleWimble [55]. Fortunately, it appears that the Bulletproofs repositories used by these applications implement strong Fiat-Shamir transformations, so no concrete applications are vulnerable. [2]

Because we want to understand how future applications could be broken by weak Fiat-Shamir transformations, though, we believe it is useful to perform a counterfactual case study:

[2]After this paper was accepted, we discovered another vulnerable implementation of Bulletproofs used by Incognito Chain [51], whose privacy protocol shares similarities with that of Monero. At the time of writing, the vulnerability has been patched; we will defer a full writeup to a future version of our paper.

what if an implementation of the MimbleWimble protocol had used a vulnerable Bulletproofs implementation?

MimbleWimble background. MimbleWimble [55] is a cryptocurrency protocol that uses Bulletproofs to achieve confidential transactions. Coins are represented as Pedersen commitments to a value v and blinding factor r. Coins are spent by transactions consisting of input coins $\{C_{\text{in},1}, ..., C_{\text{in},n}\}$, output coins $\{C_{\text{out},1}, ..., C_{\text{out},m}\}$, a value S, and a "transaction kernel" consisting of different types of validity proofs. The number of input and output coins is limited in some cases to small values like 20 and 30, respectively, though Litecoin's implementation [59] could potentially allow hundreds of output coins. Among these proofs is a range proof that the value of each input and output coin is in a specified range small enough to ensure the sums $\sum_{i=1}^{n} v_{\text{in},i}$ and $\sum_{i=1}^{m} v_{\text{out},i}$ do not overflow modulo the group order p. (A typical choice for p will be roughly 256 bits.) To be a valid transaction, the equation

$$\sum_{i=1}^{n} v_{in,i} - \sum_{i=1}^{m} v_{out,i} = S \mod p \tag{3}$$

must be satisfied. An important additional constraint is that the public supply value S is relatively small — e.g., in Litecoin [59], they must be in the range $[0, 2^{64}]$.

Attacking MimbleWimble. Ordinarily, the range proofs prevent the committed coin values from being large enough to overflow mod p; however, our attack allows one to compute valid range proofs for commitments to uniformly random elements of $\mathbb{Z}_p$ (which are highly likely to be outside the range). Thus, to craft a valid transaction that forges money, an attacker need only construct output coins with values that satisfy Equation 3. The difficulty of doing this depends on whether the protocol uses the aggregate range proof for all coins, or a single range proof for each coin.

In the case where each output coin has its own range proof—and therefore each value can be chosen independently of all others—we can express this as a generalized birthday problem [88] with as many lists as there are output coins. For simplicity, assume the attacker uses 30 output coins. In expectation, as soon as each list has $2^{\lceil \log p \rceil / 30} \approx 2^9$ elements, a solution will exist, but may be difficult to compute efficiently. By applying the k-sum algorithm of [88] for $k = 30$, we can compute a solution in time roughly 2^{43} after computing roughly 2^{43} forged proofs for each of the 30 coins. (For simplicity, we ignore other choices an attacker could make, such as choosing S or some of the input coins, that might make the attack less expensive.)

Our attack relies on being able to choose each output coin's value independently of all others. If MimbleWimble used the Bulletproofs aggregate range proof protocol, our generalized birthday attack would not obviously translate, since the last step of the weak Fiat-Shamir forgery would choose all 30 output coins at the same time. Surprisingly, we show that an even easier attack is possible against MimbleWimble if an aggregate range proof is used. (See Figure 6, which shows our attack against BP-ARP instantiated with a weak Fiat-Shamir transformation.) Note in the figure that in the last step, the public input $\mathbf{V}$ is chosen by solving linear equations for the values and blinding factors. Adding in Equation 3 as another linear constraint on the values, we only have two constraints and 30 variables. Thus, the attacker can freely choose the values of 28 of the output coins, and must only set the last two so that the forged proof is valid and the balance equation holds. This attack is very fast, needing only to solve a small linear system in $\mathbb{F}$.

In both cases, once the attacker has crafted a valid transaction with forged output coins, they have created funds out of thin air by overflowing the balance equation. To spend newly-created coins whose values are outside the allowed range, the attacker would need to craft another weak F-S proof that overflows the balance equation again.

V. Plonk

A. Protocol Description

Constraint system. Plonk handles fan-in two arithmetic circuits with unlimited fan-out. For such a circuit with n gates and m wires, we define a constraint system $\mathcal{C} = (\mathcal{V}, \mathcal{Q})$ where

- $\mathcal{V} = (\mathbf{a}, \mathbf{b}, \mathbf{c}) \in ([m]^n)^3$ consists of the left, right, and output sequence.
- $\mathcal{Q} = (\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C}) \in (\mathbb{F}^n)^5$ consists of selector vectors.

Here $\mathbb{F}$ is a finite field containing a subgroup $H = \langle \omega \rangle$ of order n. An assignment of values to wires $\mathbf{x} \in \mathbb{F}^m$ satisfies $\mathcal{C}$ if

$$\mathbf{q_L} \circ \mathbf{x_a} + \mathbf{q_R} \circ \mathbf{x_b} + \mathbf{q_O} \circ \mathbf{x_c} + \mathbf{q_M} \circ \mathbf{x_a} \circ \mathbf{x_b} + \mathbf{q_C} = \mathbf{0}.$$

To define a relation $\mathcal{R}$ based on $\mathcal{C}$, we set a subset $\{1, \ldots, \ell\}$ of the wires to be public inputs PI and the rest to be the witness. The constraint system is set up so that the first ℓ constraints are of the form $\mathbf{x}_{\mathbf{a}_i} - \mathsf{PI}_i = 0$, where $\mathbf{a}_i = i$.

Converting to polynomial constraints. Plonk proves the satisfiability of its constraint system by reducing to certain polynomial identities. We encode vectors into polynomials in the Lagrange basis, i.e. we define $\mathsf{q_Y}(X) = \sum_{i=1}^{n} (\mathbf{q_Y})_i \mathsf{L}_i(X)$ for $\mathbf{Y} \in \{\mathbf{L}, \mathbf{R}, \mathbf{O}, \mathbf{M}, \mathbf{C}\}$. The public input and witness are encoded into three polynomials $\mathsf{p}(X) = \sum_{i=1}^{n} \mathbf{x}_{\mathbf{p}_i} \mathsf{L}_i(X)$ for $\mathsf{p} \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. Circuit satisfiability then reduces to checking that $\mathsf{eq}(X)$ vanishes on H, where

$$\begin{aligned}\mathsf{eq}(X) = \mathsf{a}(X)\mathsf{b}(X)\mathsf{q_M}(X) + \mathsf{a}(X)\mathsf{q_L}(X) + \mathsf{b}(X)\mathsf{q_R}(X) \\ + \mathsf{c}(X)\mathsf{q_O}(X) + \mathsf{PI}(X) + \mathsf{q_C}(X).\end{aligned}$$

We also need to check the consistency of the wiring. [37] defines a permutation $\sigma : [3n] \to [3n]$ that encodes this consistency check, converts it into polynomial constraints by letting the prover send a polynomial $\mathsf{z}(X)$, and then checks that the following holds over H: (1) $(\mathsf{z}(X) - 1)\mathsf{L}_1(X) = 0$ and (2) $\mathsf{per}(X) = 0$, where

$$\begin{aligned}\mathsf{per}(X) = &(\mathsf{a}(X) + \beta X + \gamma)(\mathsf{b}(X) + \beta k_1 X + \gamma) \\ &(\mathsf{c}(X) + \beta k_2 X + \gamma)\mathsf{z}(X) \; - \; (\mathsf{a}(X) + \beta \mathsf{S}_{\sigma 1}(X) + \gamma) \\ &(\mathsf{b}(X) + \beta \mathsf{S}_{\sigma 2}(X) + \gamma)(\mathsf{c}(X) + \beta \mathsf{S}_{\sigma 3}(X) + \gamma)\mathsf{z}(\omega X).\end{aligned}$$

Preprocessed Polynomials.

- Selector polynomials $\mathsf{q_L}(X), \mathsf{q_R}(X), \mathsf{q_O}(X), \mathsf{q_M}(X), \mathsf{q_C}(X)$.
- Permutation polynomials $(\mathsf{S}_{\sigma 1}(X), \mathsf{S}_{\sigma 2}(X), \mathsf{S}_{\sigma 3}(X))$.

Public Input. $(w_i)_{i\in[\ell]}$. **Witness.** $(w_i)_{i\in[\ell+1,3n]}$.

Interaction Phase.

1) $\mathcal{P}$ samples $b_1, \ldots, b_6 \overset{\$}{\leftarrow} \mathbb{F}$ and sends wire polynomials

$$\mathsf{a}(X) = \sum_{i=1}^{n} w_i \mathsf{L}_i(X) + (b_1 X + b_2)\mathsf{Z_H}(X),$$

$$\mathsf{b}(X) = \sum_{i=1}^{n} w_{n+i} \mathsf{L}_i(X) + (b_3 X + b_4)\mathsf{Z_H}(X),$$

$$\mathsf{c}(X) = \sum_{i=1}^{n} w_{n+i} \mathsf{L}_i(X) + (b_5 X + b_6)\mathsf{Z_H}(X).$$

2) $\mathcal{V}$ sends permutation challenges $\beta, \gamma \overset{\$}{\leftarrow} \mathbb{F}$.

3) $\mathcal{P}$ samples $b_7, b_8, b_9 \overset{\$}{\leftarrow} \mathbb{F}$ and sends permutation polynomial

$$\mathsf{z}(X) = (b_7 X^2 + b_8 X + b_9)\mathsf{Z_H}(X) + \mathsf{L}_1(X) + \sum_{i=2}^{n} \mathsf{L}_i(X) \cdot$$
$$\prod_{j=1}^{i-1} \frac{(w_j + \beta\omega^{j-1} + \gamma)(w_{n+j} + \beta k_1 \omega^{j-1} + \gamma)(w_{2n+j} + \beta k_2 \omega^{j-1} + \gamma)}{(w_j + \beta\omega^*(j) + \gamma)(w_{n+j} + \beta\omega^*(n+j) + \gamma)(w_{2n+j} + \beta\omega^*(2n+j) + \gamma)}.$$

4) $\mathcal{V}$ sends quotient challenge $\alpha \overset{\$}{\leftarrow} \mathbb{F}$.

5) $\mathcal{P}$ computes quotient polynomial

$$\mathsf{t}(X) = \frac{1}{\mathsf{Z_H}(X)}\left(\mathsf{eq}(X) + \alpha\,\mathsf{per}(X) + \alpha^2(\mathsf{z}(X) - 1)\mathsf{L}_1(X)\right)$$

split into three parts

$$\mathsf{t}(X) = \mathsf{t}'_{\mathsf{lo}}(X) + X^n \mathsf{t}'_{\mathsf{mid}}(X) + X^{2n}\mathsf{t}'_{\mathsf{hi}}(X).$$

$\mathcal{P}$ then samples $b_{10}, b_{11} \overset{\$}{\leftarrow} \mathbb{F}$ and sends polynomials

$$\mathsf{t_{lo}}(X) = \mathsf{t}'_{\mathsf{lo}}(X) + b_{10}X^n,$$
$$\mathsf{t_{mid}}(X) = \mathsf{t}'_{\mathsf{mid}}(X) - b_{10} + b_{11}X^n,$$
$$\mathsf{t_{hi}}(X) = \mathsf{t}'_{\mathsf{hi}}(X) - b_{11}.$$

Query Phase.

1) $\mathcal{V}$ samples evaluation challenge $\mathfrak{z} \overset{\$}{\leftarrow} \mathbb{F}$.

2) $\mathcal{V}$ queries the polynomial oracles $\mathsf{q_Y}(X)$ for $\mathsf{Y} \in \{\mathsf{L}, \mathsf{R}, \mathsf{O}, \mathsf{M}, \mathsf{C}\}$, $\mathsf{S}_{\sigma j}(X)$ for $j \in \{1,2,3\}$, $\mathsf{a}(X)$, $\mathsf{b}(X)$, $\mathsf{c}(X)$, $\mathsf{z}(X)$, $\mathsf{t_{lo}}(X)$, $\mathsf{t_{mid}}(X)$, $\mathsf{t_{hi}}(X)$ at $\mathfrak{z}$, and $\mathsf{z}(X)$ at $\mathfrak{z}\omega$. $\mathcal{V}$ receives the corresponding evaluations from $\mathcal{P}$.

3) $\mathcal{V}$ computes $\mathsf{Z_H}(\mathfrak{z}) = \mathfrak{z}^n - 1$, $\mathsf{L}_1(\mathfrak{z}) = \frac{\omega(\mathfrak{z}^n - 1)}{n(\mathfrak{z} - \omega)}$, and $\mathsf{PI}(\mathfrak{z}) = \sum_{i\in[\ell]} w_i \mathsf{L}_i(\mathfrak{z})$.

4) $\mathcal{V}$ uses the above evaluations to check that Equation 4 holds at $\mathfrak{z}$, namely that

$$\mathsf{eq}(\mathfrak{z}) + \alpha \cdot \mathsf{per}(\mathfrak{z}) + \alpha^2 \cdot (\mathsf{z}(\mathfrak{z}) - 1)\mathsf{L}_1(\mathfrak{z})$$
$$= \mathsf{Z_H}(\mathfrak{z})(\mathsf{t_{lo}}(\mathfrak{z}) + \mathfrak{z}^n \mathsf{t_{mid}}(\mathfrak{z}) + \mathfrak{z}^{2n}\mathsf{t_{hi}}(\mathfrak{z})).$$

Fig. 7: The Plonk Polynomial IOP

Here $\mathsf{S}_{\sigma j}(X)$ are uniquely defined based on σ for $j \in \{1,2,3\}$, k_1, k_2 are chosen such that $H \neq k_1 H \neq k_2 H$, and β, γ are the verifier's challenges. The three vanishing claims over H can be batched together with a challenge α, and by the prover sending a quotient polynomial $\mathsf{t}(X)$ satisfying

$$\mathsf{eq}(X) + \alpha \cdot \mathsf{per}(X) + \alpha^2 \cdot (\mathsf{z}(X) - 1)\mathsf{L}_1(X) = \mathsf{Z_H}(X)\mathsf{t}(X), \quad (4)$$

where $\mathsf{Z_H}(X) = \prod_{h\in H}(X - h)$.

The Plonk **polynomial IOP.** We now describe Plonk as a polynomial IOP; see Figure 7 for the full protocol. By a slight abuse of notation, we use Plonk (and later Spartan) to refer to both the polynomial IOP and the interactive argument obtained after instantiating with a polynomial commitment scheme; the usage will be clear from context. The preprocessed polynomials consist of the selector polynomials $\mathsf{q_Y}(X)$ for $\mathsf{Y} \in \{\mathsf{L}, \mathsf{R}, \mathsf{O}, \mathsf{M}, \mathsf{C}\}$, and the polynomials $\mathsf{S}_{\sigma j}(X)$ for $j \in \{1,2,3\}$. In the first round, the prover $\mathcal{P}$ sends polynomials $\mathsf{a}(X)$, $\mathsf{b}(X)$, $\mathsf{c}(X)$ encoding the public input and witness. The verifier $\mathcal{V}$ responds with challenges $\beta, \gamma \overset{\$}{\leftarrow} \mathbb{F}$ used in the permutation argument. In the second round, $\mathcal{P}$ sends the permutation polynomial $\mathsf{z}(X)$, and $\mathcal{V}$ responds with challenge $\alpha \overset{\$}{\leftarrow} \mathbb{F}$ used in batching the polynomial checks. In the third round, $\mathcal{P}$ sends the quotient polynomial $\mathsf{t}(X)$, broken down into three parts $\mathsf{t_{lo}}(X)$, $\mathsf{t_{mid}}(X)$, $\mathsf{t_{hi}}(X)$ of small degree to be compatible with the polynomial commitment scheme. (To achieve zero-knowledge, all the polynomials sent here by $\mathcal{P}$ are blinded.) In the query phase, $\mathcal{V}$ samples an evaluation point $\mathfrak{z} \overset{\$}{\leftarrow} \mathbb{F}$, then checks the polynomial identity (4) at $\mathfrak{z}$ by querying all polynomials sent by $\mathcal{P}$ at that point.

Our exposition differs from [37], which described Plonk with certain optimizations specific to the KZG polynomial commitment scheme [56]. This does not affect the applicability of our attack, as discussed next.

B. Attack Explanation

We consider the weak Fiat-Shamir variant $\mathsf{Plonk_{wFS}}$, which is the non-interactive argument obtained by applying the transformation in Section II-F to the Plonk polynomial IOP. Our attack is presented in Figure 8; there, we assume that Plonk is instantiated with a polynomial commitment scheme supporting polynomials of degree up to $n + 5$, and denote by $[\mathsf{p}]$ the commitment to a polynomial $\mathsf{p}(X)$. Since the public input PI is not bound to the challenges, our cheating prover will do the following: (1) in the first three rounds, send commitments to arbitrarily chosen polynomials, (2) provide a proof of correct evaluations for all polynomials at the challenge point, and (3) set the public input PI to satisfy the verifier's check.

Specializing our attack to [37]. The Plonk protocol in [37] leverages the homomorphic property of KZG to make the following optimizations. First, the prover only needs to send evaluations $(\mathsf{a}(\mathfrak{z}), \mathsf{b}(\mathfrak{z}), \mathsf{c}(\mathfrak{z}), \mathsf{S}_{\sigma 1}(\mathfrak{z}), \mathsf{S}_{\sigma 2}(\mathfrak{z}), \mathsf{z}(\omega\mathfrak{z}))$, and the verifier will homomorphically compute a commitment to the linearized polynomial $\mathsf{r}(X)$. Instead of checking that Equation 4 holds at $\mathfrak{z}$, the verifier only needs to check $\mathsf{r}(\mathfrak{z}) = 0$. Second, the evaluation checks can be batched together with challenges $u, v \overset{\$}{\leftarrow} \mathbb{F}$. Our attack easily specializes to these optimizations, with the only change in step 7. In that step, the malicious prover will compute evaluations $(\mathsf{a}(\mathfrak{z}), \mathsf{b}(\mathfrak{z}), \mathsf{c}(\mathfrak{z}), \mathsf{S}_{\sigma 1}(\mathfrak{z}), \mathsf{S}_{\sigma 2}(\mathfrak{z}), \mathsf{z}(\omega\mathfrak{z}))$ and append them to π, query challenge $v \leftarrow \mathsf{H}(\pi)$, then compute a batched proof of correct evaluations and append it to π.

0) Initialize empty proof $\pi = \epsilon$. Compute preprocessed polynomials $\mathsf{q_Y}(X)$ for $\mathsf{Y} \in \{\mathsf{L,R,O,M,C}\}$ and $\mathsf{S}_{\sigma \mathsf{j}}(X)$ for $j \in \{1,2,3\}$. Compute their commitments $\{[\mathsf{q_Y}]\}$, $\{[\mathsf{S}_{\sigma \mathsf{j}}]\}$ and append them to pp.

1) Choose arbitrary polynomials $\mathsf{a}(X)$, $\mathsf{b}(X)$, $\mathsf{c}(X) \in \mathbb{F}^{<n}[X]$. Compute $[\mathsf{a}], [\mathsf{b}], [\mathsf{c}]$ and append them to π.

2) Query challenges $\beta, \gamma \leftarrow \mathsf{H}(\pi)$.

3) Choose an arbitrary polynomial $\mathsf{z}(X) \in \mathbb{F}^{<n}[X]$. Compute $[\mathsf{z}]$ and append it to π.

4) Query challenge $\alpha \leftarrow \mathsf{H}(\pi)$.

5) Choose arbitrary polynomials $\mathsf{t_{lo}}(X), \mathsf{t_{mid}}(X) \in \mathbb{F}^{<n}[X]$ and $\mathsf{t_{hi}}(X) \in \mathbb{F}^{<n+5}[X]$. Compute $[\mathsf{t_{lo}}]$, $[\mathsf{t_{mi}}]$, $[\mathsf{t_{hi}}]$ and append them to π.

6) Query challenge $\mathfrak{z} \leftarrow \mathsf{H}(\pi)$.

7) Compute evaluations at $\mathfrak{z}$ of polynomials $\{\mathsf{q_Y}(X)\}_{\mathsf{Y}\in\{\mathsf{L,R,O,M,C}\}}$, $\{\mathsf{S}_{\sigma j}(X)\}_{j\in[3]}$, $\mathsf{a}(X)$, $\mathsf{b}(X)$, $\mathsf{c}(X)$, $\mathsf{z}(X)$, $\mathsf{z}(\omega X)$, $\mathsf{t_{lo}}(X)$, $\mathsf{t_{mid}}(X)$, $\mathsf{t_{hi}}(X)$, along with proofs of correct evaluations. Append evaluations and their proofs to π.

8) Set the public input $\mathsf{PI} \in \mathbb{F}^{\ell}$ to satisfy the equation

$$\sum_{i=1}^{\ell} \mathsf{PI}_i \cdot \mathsf{L}_i(\mathfrak{z}) = \mathsf{Z_H}(\mathfrak{z}) \left(\mathsf{t_{lo}}(\mathfrak{z}) + \mathfrak{z}^n \mathsf{t_{mid}}(\mathfrak{z}) + \mathfrak{z}^{2n}\mathsf{t_{hi}}(\mathfrak{z})\right) - \mathsf{eq}'(\mathfrak{z}) - \alpha \cdot \mathsf{per}(\mathfrak{z}) - \alpha^2 \cdot (\mathsf{z}(\mathfrak{z}) - 1)\mathsf{L}_1(\mathfrak{z}), \quad (5)$$

where

$$\mathsf{eq}'(\mathfrak{z}) = \mathsf{a}(\mathfrak{z})\mathsf{b}(\mathfrak{z})\mathsf{q_M}(\mathfrak{z}) + \mathsf{a}(\mathfrak{z})\mathsf{q_L}(\mathfrak{z}) + \mathsf{b}(\mathfrak{z})\mathsf{q_R}(\mathfrak{z}) + \mathsf{c}(\mathfrak{z})\mathsf{q_O}(\mathfrak{z}) + \mathsf{q_C}(\mathfrak{z}).$$

9) Output (PI, π).

Fig. 8: Weak Fiat-Shamir Attack Against $\mathsf{Plonk_{wFS}}$

Efficiency. Asymptotically, our attack runs in time $O(n)$, which is faster than the $O(n \log n)$ time of generating honest proofs due to the use of FFTs. When $\ell \geq 2$, given a proof generated according to our attack, one can reuse the same proof for different choices of PI as long as PI is chosen to satisfy Equation 5.

We implemented our attack in 300 lines of JavaScript and verified our proofs are accepted by snarkjs. We benchmarked the forged proof generation on the same machine as our Bulletproofs attack in Section IV. Our implementation was able to generate proofs for constraint systems of size 256 in 5167 milliseconds and for constraint systems of size 2048 in 8057 milliseconds.

Provable insecurity. We show that our attack breaks the knowledge soundness of $\mathsf{Plonk_{wFS}}$, assuming the Plonk relation $\mathcal{R}$ satisfies a variant of worst-case hardness.

Definition 5: *A relation* $\mathcal{R} \subseteq \mathbb{F}^{\ell} \times \mathbb{F}^{n}$ *satisfies* all-but-one (worst-case) hardness (ABO-H) *if there exists* $i \in [\ell]$ *and* $\mathsf{PI}[\,\hat{i}\,] \in \mathbb{F}^{\ell-1}$ *such that for all PPT adversaries* $\mathcal{A}$*, the following probability is negligible in* λ*:*

$$\mathbf{Adv}^{\mathsf{ABO\text{-}H}}_{\mathcal{R}}(\mathcal{A}) := \Pr\left[(\mathsf{PI}, w) \in \mathcal{R} \mid (\mathsf{PI}_i, w) \leftarrow \mathcal{A}(\mathsf{PI}[\,\hat{i}\,])\right].$$

Here $\mathsf{PI}[\,\hat{i}\,]$ *denotes the public input without the* i*th entry.*

We briefly comment on the strength of this hardness notion. If $\mathcal{R}$ is not hard in the worst case, any proof system for $\mathcal{R}$ trivially satisfies knowledge soundness, since there exists a PPT extractor $\mathcal{E}$ that brute forces the witness from any public input. Therefore, worst-case hardness is a *necessary* assumption; our notion is slightly stronger than the worst-case hardness for $\mathcal{R}$ by requiring that for some $i \in [\ell]$, worst-case hardness holds for a related relation $\mathcal{R}_i$ that puts PI_i as part of the witness instead of the public input. We expect ABO-H to hold for many relations in practice, such as the relation for the pre-image of a hash.

Theorem 6: *Assume the* Plonk *relation* $\mathcal{R}$ *satisfies* ABO-H. *Then* $\mathsf{Plonk_{wFS}}$ *is not knowledge sound.*

The intuition for the proof is as follows. Note that for any $i \in [\ell]$ and $x \in \mathbb{F}^{\ell-1}$, the malicious prover in our attack can construct an accepting proof with $\mathsf{PI}[\,\hat{i}\,] = x$. Thus, an extractor would have to find a witness for that choice of $\mathsf{PI}[\,\hat{i}\,]$, which breaks the ABO-H property of $\mathcal{R}$.

Proof: Assume that the Plonk relation $\mathcal{R}$ satisfies ABO-H, and let $i \in [\ell]$, $x \in \mathbb{F}^{\ell-1}$ be the hard instance for $\mathcal{R}$. Denote by $\mathcal{P}^*$ the malicious prover for the attack described in Figure 8, with the following specification for step 8: $\mathcal{P}^*$ sets $\mathsf{PI}[\,\hat{i}\,] = x$, then computes the unique value of PI_i that satisfies Equation 5. We will show that for every extractor $\mathcal{E}$, there exists an adversary $\mathcal{A}$, nearly as efficient as $\mathcal{E}$, against ABO-H of $\mathcal{R}$ such that

$$\Pr\left[\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\mathsf{Plonk_{wFS}},\mathcal{R}}\right] \leq \mathbf{Adv}^{\mathsf{ABO\text{-}H}}_{\mathcal{R}}(\mathcal{A}).$$

Similar to the above proof, this will imply that $\mathsf{Plonk_{wFS}}$ is not knowledge sound. The adversary $\mathcal{A}$ works as follows. $\mathcal{A}$ receives x from the ABO-H game. $\mathcal{A}$ then computes the public paramters pp (which includes the preprocessed polynomial commitments), and runs $\mathcal{P}^*$ once to get a pair (PI, π) with $\mathsf{PI}[\,\hat{i}\,] = x$. It then sends $(\mathcal{P}^*, \mathsf{pp}, \mathsf{PI}, \pi)$ to $\mathcal{E}$, and once $\mathcal{E}$ returns a witness w, $\mathcal{A}$ outputs (PI, w). We can easily see that if game $\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\mathsf{Plonk_{wFS}},\mathcal{R}}$ outputs 1, then $\mathcal{E}$ outputs a valid witness; hence, $\mathcal{A}$ wins the ABO-H game as well.

C. Practical Impacts

Affected implementations. We surveyed 12 implementations of Plonk to determine if they were vulnerable to an attack against their Fiat-Shamir transformations. Of the 12 implementations surveyed, we found 5 to be vulnerable, 4 of which have now been fixed. To understand how our attack on the soundness of Plonk could impact applications using vulnerable implementations, we investigate the application of one of the vulnerable implementations we found: Dusk Network. We selected Dusk because it is a nontrivial application of Plonk for which we found a vulnerable implementation, and it's fairly well-documented. The Dusk Network protocol is currently in its Daylight Testnet launch with a full deployment of the protocol on the roadmap for the near future.

Dusk Network background. Dusk Network is a privacy-preserving distributed ledger protocol [32]. It uses a UTXO-based transaction model called Phoenix, which works over "notes" stored on the ledger. (We will explain only the details relevant for our attack.) For simplicity, we can think of each note as being a commitment to a value. A transaction is defined by a set of input notes $(\text{in}_1, \ldots, \text{in}_m)$, a set of (newly-created) output notes $(\text{out}_1, \ldots, \text{out}_n)$, and a zero-knowledge proof of correctness π generated using Plonk. To spend each input note, the payer must reveal its *nullifier*—a random, unique identifier of an input that prevents double-spending, but does not reveal the input itself. In Dusk, the nullifier is computed as a hash of the note's index in the Merkle tree and the opening of its commitment.

The transaction circuit (described in more detail in [30], §3) takes as public inputs the Merkle root, the output notes, and the nullifiers of the input notes. It takes as private input the input notes, their openings, their Merkle paths, and the openings of the output notes. It verifies that each nullifier corresponds to a valid input note, each input note has a valid path to the Merkle root, the output commitment is well-formed, and that the sums of the values of the inputs and outputs are equal.

Attacking Dusk Network. Our attack on $\mathsf{Plonk}_{\mathsf{wFS}}$'s weak Fiat-Shamir transformation lets us obtain a satisfying proof for an arbitrary witness by setting the public input to be a specific value that will satisfy the verifier's check. Notably, our attack actually allows for an attacker to set all but one of the public inputs to arbitrary values; the last public input must be set to the (unique) value that causes the verifier's check to pass. To use our attack against Dusk, then, we must make sure that there is some public input that can be set arbitrarily and is not checked elsewhere in the protocol. We observe that the nullifier is a natural choice for this—its only external constraint is a check that it has not been used by any previous transaction; further, by design it is a random-looking value.

Thus, an attacker can use our attack to create verifying transactions that do not satisfy the circuit's constraints. One clear way to exploit this is to steal funds from the Dusk Network: an attacker can create a transaction that spends one input and sends outputs with arbitrarily large values to themselves. Because the output coins can be chosen freely, the attacker can ensure the output notes are well-formed and can be later traded for other coins.

VI. Spartan

A. Protocol Description

Constraint system. Spartan proves the satisfiability of rank-one constraint systems (R1CS). A R1CS relation is defined by a tuple $(\mathbb{F}, A, B, C, m, n, \ell)$ where $A, B, C \in \mathbb{F}^{m\times m}$ are matrices, each with at most $n = \Omega(m)$ non-zero entries, and $m \geq \ell + 1$. Given a R1CS public input $\mathsf{PI} \in \mathbb{F}^{\ell}$, a R1CS witness is a vector $w \in \mathbb{F}^{m-\ell-1}$ such that if $Z = (\mathsf{PI}, 1, w)$, then $(A \cdot Z) \circ (B \cdot Z) = C \cdot Z$. Spartan also requires that $m = 2^{\mu}$ is a power of two, and $\ell = m/2 - 1$.

Converting to polynomial constraints. We interpret the matrices A, B, C as functions from $\{0,1\}^{\mu} \times \{0,1\}^{\mu}$ to $\mathbb{F}$, and similarly $Z : \{0,1\}^{\mu} \to \mathbb{F}$, by writing the indices as their binary representation. We then consider the multilinear extensions $\widetilde{A}, \widetilde{B}, \widetilde{C}, \widetilde{Z}$ of these functions, and define the polynomial

$$\mathcal{F}_{\mathsf{PI}}(X) = \overline{A}(X) \cdot \overline{B}(X) - \overline{C}(X),$$

where $\overline{M}(X) = \sum_{y \leftarrow \{0,1\}^{\mu}} \widetilde{M}(X, y) \cdot \widetilde{Z}(y)$ for

Protocol Notation.

$$e \leftarrow \langle \mathcal{P}_{\mathsf{SC}}(p), \mathcal{V}_{\mathsf{SC}}(r)\rangle(\mu, d, T),$$

where $p \in \mathbb{F}^{\leq d}[X_1, \ldots, X_{\mu}]$ satisfies $\sum_{x\in\{0,1\}^{\mu}} g(x) = T$, and $r = (r_1, \ldots, r_{\mu}) \in \mathbb{F}^{\mu}$ is $\mathcal{V}_{\mathsf{SC}}$'s randomness.

Interaction Phase. For $i = 1, \ldots, \mu$:

1) $\mathcal{P}$ computes and sends

$$p_i(X) = \sum_{x_{i+1},\ldots,x_{\mu}\in\{0,1\}} p(r_1, \ldots, r_{i-1}, X, x_{i+1}, \ldots, x_{\mu}).$$

2) $\mathcal{V}$ sends $r_i \xleftarrow{\$} \mathbb{F}$.

Query Phase.

1) $\mathcal{V}$ checks that $p_1(0) + p_1(1) = T$.
2) $\mathcal{V}$ checks that $p_i(0) + p_i(1) = p_{i-1}(r_{i-1})$ for $2 \leq i \leq \mu$.

Output. $\mathcal{P}$, $\mathcal{V}$ outputs $e = p_{\mu}(r_{\mu})$ supposedly equal to $p(r_1, \ldots, r_{\mu})$.

Fig. 9: The Sumcheck Protocol

Preprocessed Polynomials. Multilinear extensions $\widetilde{A}(X,Y)$, $\widetilde{B}(X,Y)$, $\widetilde{C}(X,Y)$ of R1CS matrices $A, B, C \in \mathbb{F}^{m\times m}$.

Public Input. $\mathsf{PI} \in \mathbb{F}^{m/2-1}$. **Witness.** $w \in \mathbb{F}^{m/2}$.

Interaction Phase. Let $\mu = \log m$.

1) $\mathcal{P}$ sends the multilinear extension $\widetilde{w}$ of the witness w.
2) $\mathcal{V}$ sends challenge $\tau \xleftarrow{\$} \mathbb{F}^{\mu}$.
3) $\mathcal{P}$ and $\mathcal{V}$ engage in a sumcheck protocol for

$$e_x \leftarrow \langle \mathcal{P}_{\mathsf{SC}}(\mathcal{G}_{\mathsf{PI},\tau}), \mathcal{V}_{\mathsf{SC}}(r_x)\rangle(\mu, 3, 0),$$

where $\mathcal{G}_{\mathsf{PI},\tau}(X)$ is defined as in Equation 6.

4) $\mathcal{P}$ computes $v_A = \overline{A}(r_x)$, $v_B = \overline{B}(r_x)$, $v_C = \overline{C}(r_x)$ and sends (v_A, v_B, v_C) to $\mathcal{V}$.
5) $\mathcal{V}$ sends challenges $r_A, r_B, r_C \xleftarrow{\$} \mathbb{F}$.
6) $\mathcal{P}$ and $\mathcal{V}$ engage in another sumcheck protocol for

$$e_y \leftarrow \langle \mathcal{P}_{\mathsf{SC}}(\mathcal{H}_{r_x}), \mathcal{V}_{\mathsf{SC}}(r_y)\rangle(\mu, 2, T),$$

where $\mathcal{H}_{r_x}(Y)$ and T are defined as in Equation 7.

Query Phase.

1) Reject if either of the sumcheck instances fail.
2) Check that $e_x \overset{?}{=} (v_A \cdot v_B - v_C) \cdot \widetilde{eq}(r_x, \tau)$.
3) Query $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at (r_x, r_y) and receive evaluations v_1, v_2, v_3 respectively.
4) Query $\widetilde{w}$ at $(r_y)_{[1:]}$ and receive evaluation v_w.
5) Check that $e_y \overset{?}{=} (r_A \cdot v_1 + r_B \cdot v_2 + r_C \cdot v_3) \cdot v_Z$, where

$$v_Z = \Big((r_y)_0 \cdot \widetilde{(\mathsf{PI}, 1)}((r_y)_{[1:]}) + (1 - (r_y)_0) \cdot v_w \Big).$$

Fig. 10: The Spartan Polynomial IOP

$M \in \{A,B,C\}$. Note that $\mathcal{F}_{\mathsf{PI}}(X)$ vanishes on $\{0,1\}^\mu$ if and only if Z satisfies the R1CS relation. We turn this vanishing condition into a sumcheck instance by defining $\mathcal{G}_{\mathsf{PI},\tau}(X) = \mathcal{F}_{\mathsf{PI}}(X) \cdot \widetilde{\mathsf{eq}}(X,\tau)$ for a random $\tau \in \mathbb{F}^\mu$, supplied by the verifier. The goal is then to prove that

$$\sum_{x \in \{0,1\}^\mu} \mathcal{G}_{\mathsf{PI},\tau}(x) = 0. \tag{6}$$

The Spartan **polynomial IOP.** We describe the full protocol in Figure 10, which uses the sumcheck protocol described in Figure 9. The preprocessed polynomials consist of the multilinear extensions $\widetilde{A}(X,Y), \widetilde{B}(X,Y), \widetilde{C}(X,Y)$ of the R1CS matrices A,B,C. In the first round, the prover $\mathcal{P}$ sends the multilinear extension $\widetilde{w}$, and $\mathcal{V}$ sends challenge $\tau \overset{\$}{\leftarrow} \mathbb{F}^\mu$. Both parties then engage in a sumcheck protocol to prove Equation 6; after this $\mathcal{V}$ receives an evaluation e_x supposedly equal to $\mathcal{G}_{\mathsf{PI},\tau}(r_x)$, where r_x is $\mathcal{V}$'s randomness during the run of sumcheck. Since $\mathcal{V}$ cannot evaluate this itself, both parties engage in another run of sumcheck. $\mathcal{P}$ sends three values v_A, v_B, v_C supposedly equal to $\overline{A}(r_x)$, $\overline{B}(r_x)$, $\overline{C}(r_x)$ respectively, and $\mathcal{V}$ checks that

$$e_x = (v_A \cdot v_B - v_C) \cdot \widetilde{\mathsf{eq}}(r_x, \tau).$$

Next, $\mathcal{V}$ responds with three challenges $r_A, r_B, r_C \overset{\$}{\leftarrow} \mathbb{F}$ to batch three sumcheck instances into one. The second sumcheck instance is then

$$\sum_{y \in \{0,1\}^\mu} \mathcal{H}_{r_x}(y) = T, \tag{7}$$

where

$$\mathcal{H}_{r_x}(Y) = \Big(r_A \widetilde{A}(r_x, Y) + r_B \widetilde{B}(r_x, Y) + r_C \widetilde{C}(r_x, Y)\Big) \widetilde{Z}(Y),$$
$$T = r_A \cdot v_A + r_B \cdot v_B + r_C \cdot v_C.$$

After running sumcheck on Equation 7, $\mathcal{V}$ receives an evaluation e_y supposedly equal to $(r_A \widetilde{A}(r_x, r_y) + r_B \widetilde{B}(r_x, r_y) + r_C \widetilde{C}(r_x, r_y))\widetilde{Z}(r_y)$, for $\mathcal{V}$'s randomness r_y during the run of sumcheck. $\mathcal{V}$ now queries $\widetilde{A}, \widetilde{B}, \widetilde{C}$ at (r_x, r_y) for evaluations v_1, v_2, v_3, and $\widetilde{w}$ at $(r_y)_{[1:]}$ for evaluation v_w, and checks that

$$e_y = (r_A \cdot v_1 + r_B \cdot v_2 + r_C \cdot v_3) \cdot v_Z, \tag{8}$$

where $v_Z = \Big((r_y)_0 \cdot \widetilde{(\mathsf{PI},1)}((r_y)_{[1:]}) + (1-(r_y)_0) \cdot v_w\Big)$. $\mathcal{V}$ also performs checks for each sumcheck instance and rejects the results if either of the instances rejects them.

B. Attack Explanation

Figure 11 gives an attack against $\mathsf{Spartan}_{\mathsf{wFS}}$, which is the non-interactive argument obtained by applying the transformation in Section II-F to the Spartan polynomial IOP described in Figure 10. The attack is similar to the one against $\mathsf{Plonk}_{\mathsf{wFS}}$: first, the malicious prover $\mathcal{P}^*$ chooses polynomials (including witnesses) that will satisfy all verification equations except Equation 8. Then, to finish $\mathcal{P}^*$ crafts PI according to Equation 9 so as to satisfy this final check.

```
0) Compute commitments $[\widetilde{A}], [\widetilde{B}], [\widetilde{C}]$ and append them to pp.
   Initialize empty proof $\pi = \epsilon$.
1) Choose arbitrary multilinear polynomial $\widetilde{w} \in \mathbb{F}[\mu]$, compute
   $[w]$ and append it to $\pi$.
2) Query challenge $\tau \leftarrow \mathsf{H}(\pi)$.
3) For each of the two sumcheck instances:
   a) In each round $i \in [\mu]$, sample an arbitrary polynomial
      $\mathsf{p}_i(X)$ of appropriate degree that satisfies
         $\mathsf{p}_i(0) + \mathsf{p}_i(1) = \mathsf{p}_{i-1}(r_{i-1})$.
      Compute $[\mathsf{p}_i]$ and append it to $\pi$.
   b) Query challenge $r_i \leftarrow \mathsf{H}(\pi)$.
4) Between the two sumchecks, choose arbitrary $v_A, v_B, v_C \in \mathbb{F}$
   such that
         $e_x = (v_A \cdot v_B - v_C) \cdot \widetilde{\mathsf{eq}}(r_x, \tau)$.
   Query challenges $r_A, r_B, r_C \leftarrow \mathsf{H}(\pi)$.
5) Compute evaluations $v_1 = \widetilde{A}(r_x, r_y)$, $v_2 = \widetilde{B}(r_x, r_y)$,
   $v_3 = \widetilde{C}(r_x, r_y)$, $v_w = \widetilde{w}((r_y)_{[1:]})$ and valid proofs of openings.
   Append evaluations and opening proofs to $\pi$.
6) Set the public input $\mathsf{PI} \in \mathbb{F}^\ell$ to satisfy the equation
   $\widetilde{(\mathsf{PI},1)}((r_y)_{[1:]}) = (r_y)_0^{-1} \cdot (v_Z - (1-(r_y)_0) \cdot v_w)$,   (9)
   where
         $v_Z = e_y \cdot (r_A \cdot v_1 + r_B \cdot v_2 + r_C \cdot v_3)^{-1}$.
7) Output $(\mathsf{PI}, \pi)$.
```

Fig. 11: Weak Fiat-Shamir Attack Against $\mathsf{Spartan}_{\mathsf{wFS}}$

Provable insecurity. Our attack against $\mathsf{Spartan}_{\mathsf{wFS}}$ satisfies the same properties as with our attack on $\mathsf{Plonk}_{\mathsf{wFS}}$—namely that a malicious prover can arbitrarily choose all entries of PI except one. Thus, we can prove that our attack breaks the knowledge soundness of $\mathsf{Spartan}_{\mathsf{wFS}}$ assuming the same ABO-H property of the R1CS relation $\mathcal{R}$. The proof is similar to that of Theorem 6.

Theorem 7: Assume the R1CS relation $\mathcal{R}$ satisfies ABO-H. *Then* $\mathsf{Spartan}_{\mathsf{wFS}}$ *is not knowledge sound.*

Proof: Assume that the R1CS relation $\mathcal{R}$ satisfies ABO-H, and let $i \in [\ell]$, $x \in \mathbb{F}^{\ell-1}$ be the hard instance for $\mathcal{R}$. Denote by $\mathcal{P}^*$ the malicious prover for the attack described in Figure 11, with the following specification for step 8: $\mathcal{P}^*$ sets $\mathsf{PI}[\hat{i}] = x$, then computes the unique value of PI_i that satisfies Equation 9. Note that $\mathcal{P}^*$ can do this since we can write

$$\begin{aligned}\widetilde{(\mathsf{PI},1)}(r) &= \sum_{y \in \{0,1\}^{\mu-1}} (\mathsf{PI},1)(y) \cdot \widetilde{\mathsf{eq}}(r,y) \\ &= \sum_{k=1}^{m/2-2} \mathsf{PI}_k \cdot \widetilde{\mathsf{eq}}(r, \mathsf{bin}(k)) + \widetilde{\mathsf{eq}}(r, \mathsf{bin}(m/2-1)).\end{aligned}$$

Here we denote $r = (r_y)_{[1:]}$, and $\mathsf{bin}(k)$ is the binary representation of k. Since this is a linear equation in terms of PI_k's, to solve $\widetilde{(\mathsf{PI},1)}(r) = v$ for any value v, we can fix all but one entry $\mathsf{PI}[\hat{i}]$ and find a unique solution for the remaining entry PI_i.

- $\mathsf{Setup}(1^\lambda)$: Generate a finite abelian group $\mathbb{G}$ of unknown order. Pick an efficiently computable hash functions $\mathsf{H}_\mathbb{G} : \{0,1\}^* \rightarrow \mathbb{G}$ to be modeled as a random oracle. Return $\mathsf{pp} = (\mathbb{G}, \mathsf{H}_\mathbb{G})$.
- $\mathsf{Eval}(\mathsf{pp}, T, x)$:

1) Compute $y \leftarrow \mathsf{H}_\mathbb{G}(x)^{2^T}$ by repeated squaring.
2) Compute $\pi \leftarrow \mathcal{P}_{\mathsf{FS}}(\mathsf{pp}, T, x, y)$ as the F-S transformed prover of the interactive argument VDF.

- $\mathsf{Verify}(\mathsf{pp}, T, x, y, \pi)$: Return the result of the F-S verifier $\mathcal{V}_{\mathsf{FS}}(\mathsf{pp}, x, y, \pi)$.

Interactive Argument VDF:

$$\mathcal{R}_{\mathsf{VDF}} = \left\{ (\mathsf{pp}, (T, x, y), \emptyset) \mid y = \mathsf{H}_\mathbb{G}(x)^{2^T} \right\}.$$

1) $\mathcal{V}$ sends to $\mathcal{P}$ a 2λ-bit prime ℓ uniformly at random.
2) Let $g = \mathsf{H}_\mathbb{G}(x)$. $\mathcal{P}$ computes $\pi = g^{\lfloor 2^T/\ell \rfloor}$ and sends π to $\mathcal{V}$.
3) $\mathcal{V}$ computes $r = 2^T \mod \ell$ and accepts if and only if $\pi^\ell \cdot g^r = y$.

Fig. 12: Wesolowski's verifiable delay function. The Fiat-Shamir transformed argument is described in prose below.

We now show that for every extractor $\mathcal{E}$, there exists an adversary $\mathcal{A}$, nearly as efficient as $\mathcal{E}$, against ABO-H of $\mathcal{R}$ such that

$$\Pr\left[\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\mathsf{Spartan}_{\mathsf{wFS}},\mathcal{R}}\right] \leq \mathbf{Adv}^{\mathsf{ABO\text{-}H}}_{\mathcal{R}}(\mathcal{A}).$$

Similar to the above proof, this will imply that $\mathsf{Spartan}_{\mathsf{wFS}}$ is not knowledge sound. The adversary $\mathcal{A}$ receives x from the ABO-H game, then computes $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ and $(\mathsf{PI}, \pi) \leftarrow \mathcal{P}^*(\mathsf{pp})$ such that $\mathsf{PI}[\hat{i}] = x$. It then sends $(\mathcal{P}^*, \mathsf{pp}, \mathsf{PI}, \pi)$ to $\mathcal{E}$, and when $\mathcal{E}$ outputs w, $\mathcal{A}$ outputs (PI, w). We can see that if the game $\mathsf{KS}^{\mathcal{E},\mathcal{P}^*}_{1,\mathsf{Spartan}_{\mathsf{wFS}},\mathcal{R}}$ returns 1, then $\mathcal{E}$ finds a valid witness w; thus, $\mathcal{A}$ wins in the ABO-H game. This proves the inequality.

C. Practical Impacts

We found two implementations of Spartan; both were vulnerable to this attack but were fixed following an initial public disclosure of our results. Interestingly, the reference implementations of two follow-ups to Spartan, Brakedown [43] and Nova [57], were also vulnerable. Our attack has no impact on applications—as far as we know, no applications currently use Spartan (or related protocols) in production. Still, our attack gives firm evidence that future applications of Spartan should use strong F-S.

VII. Wesolowski's VDF

We describe an attack against a weak Fiat-Shamir transformation in a verifiable delay function (VDF) [14] constructed by Wesolowski [90]. In Section VII-C, we discuss how our attack affects the security of vulnerable implementations in practice.

Verifiable delay functions. A VDF is a function whose output is only known after a certain time delay, and additionally comes with a proof of correct evaluation. Formally, it is a tuple of three algorithms:

- $\mathsf{Setup}(1^\lambda) \rightarrow \mathsf{pp}$ outputs public parameters,
- $\mathsf{Eval}(\mathsf{pp}, T, x) \rightarrow (y, \pi)$ evaluates the VDF with time delay T on input x, returning output y along with a proof π. Eval is required to generate y deterministically,
- $\mathsf{Verify}(\mathsf{pp}, T, x, y, \pi)$ verifies the proof.

VDFs are required to satisfy *completeness*, *soundness*, and *sequentiality*; for full definitions see e.g. [14], [15], [90]. We note one significant departure of our syntax from the syntax of previous works: we allow the time delay T to be an input to the Eval algorithm, instead of T being determined ahead of time as a parameter to Setup. We also consider an *adaptive* soundness notion for VDF, i.e. given $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$, an attacker cannot output (T, x, y, π) with $y \neq \mathsf{Eval}(\mathsf{pp}, T, x)$ that would make Verify accept the proof; previous works did not allow an attacker to choose the delay parameter. We believe that our modeling choices are closer to practice, as many applications (see Section VII-C) do afford attackers such capabilities.

A. Protocol Description

We describe the interactive argument of [90] in Figure 12. The argument uses a function $\mathsf{H}_\mathbb{G}$ that hashes bit strings into the group $\mathbb{G}$. Let $g = \mathsf{H}_\mathbb{G}(x)$. To convince the verifier it has computed y that equals g^{2^T}, the prover (implicitly) begins by sending y to the verifier. Then, the verifier samples a random prime ℓ of 2λ bits, where λ is the security parameter, and sends ℓ to the prover. The prover replies with the value $\pi = g^{\lfloor 2^T/\ell \rfloor}$; finally, the verifier computes the residue $r = 2^T \mod \ell$ and accepts if $\pi^\ell g^r = y$. Applying the Fiat-Shamir transformation to this argument entails deriving ℓ by hashing the prover's first message with a hash function $\mathsf{H}_{\mathsf{prime}}$ that outputs 2λ-bit primes. The paper specifies the exact transformation to be used as $\ell = H_{\mathsf{prime}}(g, y)$. We call the resulting non-interactive argument $\mathsf{VDF}_{\mathsf{wFS}}$.

B. Attack Explanation

We observe that the paper [90] specified a F-S transformation that leaves out several parameters, such as the time delay T and the group description $\mathbb{G}$; thus, the paper is recommending weak F-S. This allows us to break the adaptive soundness of the VDF (defined above); our attack is presented in Figure 13. In our attack, the malicious prover first computes a legitimate proof for a small time delay t. Then, because this proof does not depend on t, the prover will choose a much larger delay T that leads to the verifier computing the same r value in the last step. The proof will still verify for the larger delay T, though the prover only did t sequential squarings. By our choice of T, with high probability we will have $y \neq \mathsf{H}_\mathbb{G}(x)^{2^T}$; otherwise, we know that $\mathsf{H}_\mathbb{G}(x)^{2^T - 2^t} = 1$, which allows us to deduce the group order of $\mathbb{G}$, breaking the low order assumption (as stated in [15]). We summarize with the following theorem.

Theorem 8: $\mathsf{VDF}_{\mathsf{wFS}}$ *does not satisfy adaptive soundness.*

1) Pick a small time delay t. For an arbitrary $x \leftarrow \{0,1\}^*$, compute $y = g^{2^t}$ where $g = \mathsf{H}_{\mathbb{G}}(x)$.
2) Compute the proof $\pi \leftarrow \mathcal{P}_{\mathsf{FS}}(\mathsf{pp}, T, x, y)$, which in particular is equal to $\pi = g^{\lfloor 2^t/\ell \rfloor}$ where $\ell = \mathsf{H}_{\mathsf{prime}}(g, y)$.
3) Pick a large T such that $2^T \equiv 2^t \mod \ell$. In particular, we can pick $T = t + \ell - 1$. More generally, we can pick $T = t + o$ where o is the order of 2 modulo ℓ.
4) Output $((T, x, y), \pi)$.

Fig. 13: Weak Fiat-Shamir Attack Against $\mathsf{VDF}_{\mathsf{wFS}}$

Note that we are not claiming the soundness result in [90] is incorrect, merely that it implicitly assumes the delay parameter is fixed.

C. Practical Impacts

Affected implementations. To assess the practical impacts of our attack against the VDF's weak Fiat-Shamir Transformation, we first checked if any implementations use weak F-S. We found that every implementation of Wesolowski's VDF we checked implemented weak F-S. We suspect this is because the paper [90] explicitly recommends weak F-S.

On the adaptivity of attackers in choosing T. Next, we looked at the applications that use vulnerable implementations. We found that the dominant use of VDFs in practice are in cryptocurrency protocols for some kind of proof of work—for example, the Chia protocol uses VDFs to let miners prove they have reserved some amount of storage space for some amount of time. These protocols allow the delay to change dynamically, depending on the state of the chain; thus, an attacker could still (in principle) influence the chosen delay.

Further constraints in practice. Our attack is theoretically possible, but turns out not to affect most implementations because of a small, but consequential, implementation choice: the data type used for the delay parameter T is often much too small to fit a delay parameter chosen by our malicious prover. For example, in Chia, the delay parameter is a 64-bit integer, but our malicious prover's delay parameter will be roughly 256 bits, unless 2 has small order modulo the challenge prime ℓ. We suspect that this happens with very small probability; assuming the order of 2 modulo ℓ is uniformly distributed, the probability of choosing ℓ that gives such a small order is about 2^{-192}.

Nevertheless, for implementation choices that allow the time delay to be up to 256 bits, our attack is realizable. Such is the case for two VDF verifiers written in Solidity and Python [1]: Solidity's default integer type is 256 bits, and Python does not have a priori bounds on its integers. We developed a proof-of-concept exploit for those verifiers; forged proof generation takes less than a second.

In summary, our attack only leads to a latent vulnerability for applications where the delay parameter T is constrained to be much smaller than the challenge prime ℓ. An interesting question for future work is whether it is possible to prove adaptive soundness for $\mathsf{VDF}_{\mathsf{wFS}}$ when this condition is enforced. Still, we believe strong F-S (i.e., hashing all public information, including the delay parameter and the group description) is the right choice for implementations.

VIII. Discussion

In this section, we discuss some general points related to our attacks. We discuss whether our attacks could be detected, document other kinds of broken F-S implementations we found, and study one case in more detail.

Detection of weak F-S attacks. Understanding how detectable our attacks are in practice requires answering two related questions. First, do forged public inputs have the same distribution as real ones? And second, do our proofs have the same distribution as honestly-generated ones?

Our attacks rely on choosing part of the public inputs as a function of the proof; thus, the public inputs output by our attacks do not necessarily have the same distribution as real ones. For Bulletproofs, the public input is a perfectly hiding commitment in both the real case and for our forger. The public inputs of our Wesolowski attack seem easily detectable, since actually performing $\approx 2^{256}$ squarings in an RSA group—as our forged proofs show the prover did—would be virtually impossible. For Plonk and Spartan, only one public input is chosen as a function of the proof and the other public inputs; intuitively, this input looks like a uniformly random field element. Reasoning about whether this is detectable is difficult, since it is highly contextual.

The proofs output by our attacks have the same distribution as honest ones in some cases, but not others: e.g., our forged Plonk proofs consist of hiding commitments to polynomials and their evaluations; the hiding property guarantees the distribution is the same as an honest prover. In contrast, though, our attack on Wesolowski's VDF outputs proofs that are distinguishable from honest ones, since they prove false statements—any party that computes the real VDF output can tell our claimed value is not correct.

In cases where our public inputs and forged proofs have the right distribution, any detection of attacks against weak F-S will have to rely on outside heuristics; for instance, monitoring the public supply in, e.g. MimbleWimble, could help detect if funds are being stolen through such an attack. Determining who is responsible for the attack would likely be more difficult.

Other misuses of Fiat-Shamir. Our implementation survey uncovered other kinds of F-S mistakes:

1) Not including one (or more) of the prover's messages in the hash computation.
2) Initializing a new transcript when invoking the prover/verifier for a subprotocol.
3) Not including all public parameters (e.g., R1CS matrices or group generators) in the hash computation.

Case 2 can be thought of as a special case of case 1; by initalizing a new transcript, one effectively excludes the prover's messages from earlier in the protocol. Both cases 1 and 2 trivially lead to soundness attacks, even in the non-adaptive case.

For case 3, the impact is going to depend on whether the public parameters are fixed, or could be attacker-chosen in some cases. Some public parameters, like generators for cyclic groups, are nearly always hard-coded and so may not need to be hashed. However, a public parameter that could be attacker-chosen is the circuit/R1CS representation for a proof system like Plonk or Spartan. If the verifier accepts arbitary circuits from a prover, and this circuit is not included in the Fiat-Shamir computation, then this can be abused. We wish to highlight this as a case deserving further study, since in many emerging applications of proof systems (such as private smart contract platforms like Aleo [5]), user-specified circuits that represent arbitrary programs are a feature of the application.

IX. Mitigating Weak F-S

In this section, we suggest how academic researchers can clarify the evident confusion about the correct use of F-S. We also suggest designs for tools that can detect weak F-S implementations programmatically, and make it easier to implement F-S correctly.

Suggestions for researchers. In reading recent papers about proof systems that use F-S, we noticed a clear pattern that may explain why confusion is so widespread. Most papers present and analyze the interactive version of the protocol, then state that F-S can be used to make the protocol non-interactive, but without specifying how this should be done, or giving too little information. An example is simply stating the "transcript" should be hashed, without saying what the transcript includes.

We suggest that, to minimize misconceptions and possibilities for error, researchers who present new protocols as interactive should be very precise about the way F-S should be applied to render their protocol non-interactive. Ideally, this includes explicitly identifying the public parameters, inputs, and prover messages that should be hashed, and specifying how to hash them.

This is not a perfect solution, since misunderstandings exist even amongst researchers: the few papers that attempt to be prescriptive about the exact transformation sometimes even state it incorrectly. For example, the original versions of both the Bulletproofs and Wesolowski's VDF papers explicitly recommend weak F-S. (We notified the authors; the Bulletproofs paper has since been updated.)

A. Automated Tooling

We explore some programmatic solutions that can either detect the incorrect use of the Fiat-Shamir transformation, or help the programmer in implementing the transformation correctly.

Criteria. We identify four key criteria to evaluate our tooling proposals, as well as any existing tooling for Fiat-Shamir, in the context of reducing weak F-S vulnerabilities.

1) *Correctness*: for detection, the tool should have a low error rate, and for implementation, the tool should result in correct implementations of Fiat-Shamir,
2) *Simplicity*: the tool should be easy to use, requiring minimal modification to the pracitioner's workflow,
3) *Misuse-resistance*: it should be difficult to use the tool in incorrect or unintended ways,
4) *Efficiency*: the tool should add negligible overhead to the runtime of the proof system.

For existing tooling, we are aware of the Merlin library [28] that implements a Transcript object with two operations: one for adding messages and one for deriving challenges. The library provides support for domain separation, message framing, and protocol composition; it has been used in many proof system libraries written in Rust. However, despite its intentional design, Merlin does not enforce the correct use of Fiat-Shamir, and indeed many of the weak Fiat-Shamir implementations we found used Merlin.

We present a few different ideas for discouraging and detecting incorrect Fiat-Shamir usage. First, Merlin could be extended to have an explicit function for adding in the public statement to the transcript. If the user does not call this function, Merlin can raise a warning alerting the user to a potential weak F-S attack. Although this would not automatically prevent incorrect instances of Fiat-Shamir, we believe it would reduce the likelihood of users missing these public values, which were the most common implementation mistake we found.

In addition to the above measure for discouraging misuse, we've also begun implementing an extension to Merlin that requires developers to specify all Fiat-Shamir inputs and challenges when the transcript is initialized. Generating challenges without providing all the required inputs will result in an error, alerting developers to potential weak Fiat-Shamir transformations. Additionally, explicitly listing the Fiat-Shamir inputs and challenges encourages developers to carefully consider Fiat-Shamir requirements.

For detection, we can utilize information flow analyses to determine which objects flow to both the proof and verification result (either directly or indirectly). We can compare these objects to those passed to the transcript. If there is a mismatch, then it is likely that Fiat-Shamir is implemented incorrectly. Since the tool acts as a plug-in during testing, it adds zero overhead (efficiency), and ideally only requires few changes to be integrated (simplicity and misuse-resistance). However, this approach suffers from a false negative rate, as it would not be able to check whether two objects are equal, even though they might be computed differently, i.e. the prover computes a proof element, while the verifier receives such a proof element. Detecting when these values are missing from the Fiat-Shamir computation would require dynamic equality checking on values shared between the prover and the verifier. Even though this approach would reduce the false negative rate, it would increase the false positive rate, as many of these shared values will not be required for Fiat-Shamir.

Acknowledgements

The authors thank Trail of Bits, Ian Smith, Riad Wahby, Fraser Brown, and the anonymous reviewers at IEEE S&P

2023 for their helpful comments and suggestions. This research was supported by DARPA under Agreement No. HR00112020022. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the United States Government or DARPA.

References

[1] 0x Project. A solidity implementation of a vdf verifier contract. https://github.com/0xProject/VDF, 2022.

[2] J. Abfalter. bulletproofs-js. https://github.com/jafalter/bulletproof-js, 2022.

[3] Adjoint Inc. Adjoint bulletproofs. https://github.com/sdiehl/bulletproofs, 2022.

[4] Adjoint Inc. Sonic implementation. https://github.com/adjoint-io/sonic, 2022.

[5] Aleo. https://www.aleo.org/, 2022.

[6] Anoma. Proof system with plonkup back-end proving arguments. https://github.com/anoma/plonkup/, 2022.

[7] A. Archer. Zero-knowledge proof implementation for passwords and other secrets. https://github.com/GoodiesHQ/noknow-python, 2022.

[8] Aztec Protocol. aztec connect repository. https://github.com/AztecProtocol/aztec-connect, 2022.

[9] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.

[10] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Heidelberg, May / June 2006.

[11] E. Ben-Sasson, A. Chiesa, and N. Spooner. Interactive oracle proofs. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part II*, volume 9986 of *LNCS*, pages 31–60. Springer, Heidelberg, Oct. / Nov. 2016.

[12] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In K. Fu and J. Jung, editors, *USENIX Security 2014*, pages 781–796. USENIX Association, Aug. 2014.

[13] D. Bernhard, O. Pereira, and B. Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In X. Wang and K. Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 626–643. Springer, Heidelberg, Dec. 2012.

[14] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, Aug. 2018.

[15] D. Boneh, B. Bünz, and B. Fisch. A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712, 2018. https://eprint.iacr.org/2018/712.

[16] J. Bootle, A. Chiesa, Y. Hu, and M. Orrù. Gemini: Elastic SNARKs for diverse environments. In O. Dunkelman and S. Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 427–457. Springer, Heidelberg, May / June 2022.

[17] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie. Consensys/gnark: v0.6.4, Feb. 2022.

[18] S. Bowe. Sonic. https://github.com/ebfull/sonic, 2022.

[19] Brakedown reference implementation. https://github.com/conroi/Spartan/tree/brakedown, 2022.

[20] Bulletproof range proof implementation in pure swift. https://github.com/shamatar/BulletproofSwift, 2022.

[21] A ruby implementation of bulletproofs. https://github.com/azuchi/bulletproofsrb/, 2023.

[22] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

[23] B. Bünz, B. Fisch, and A. Szepieniec. Transparent SNARKs from DARK compilers. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 677–706. Springer, Heidelberg, May 2020.

[24] M. Campanelli, C. Ganesh, H. Khoshakhlagh, and J. Siim. Impossibilities in succinct arguments: Black-box extraction and more. Cryptology ePrint Archive, Report 2022/638, 2022. https://eprint.iacr.org/2022/638.

[25] D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, Aug. 1993.

[26] Chia network. https://www.chia.net/, 2022.

[27] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. P. Ward. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 738–768. Springer, Heidelberg, May 2020.

[28] H. de Valence. Merlin: composable proof transcripts for public-coin arguments of knowledge. https://merlin.cool/, 2022.

[29] H. de Valence, C. Yun, and O. Andreev. A pure-rust implementation of bulletproofs using ristretto. https://github.com/dalek-cryptography/bulletproofs, 2022.

[30] Dusk genesis circuits. https://github.com/dusk-network/rusk/blob/master/circuits/transfer/doc/dusk-genesis-circuits.pdf, 2022.

[31] Pure rust implementation of the plonk zkproof system done by the dusk-network team. https://github.com/dusk-network/plonk, 2022.

[32] The dusk network whitepaper, version 2.0.0. https://dusk.network/uploads/dusk-whitepaper.pdf, 2019.

[33] Encoins bulletproofs. https://github.com/encryptedcoins/encoins-bulletproofs, 2023.

[34] Espresso Systems. A rust implementation of the plonk zkp system and extensions. https://github.com/EspressoSystems/jellyfish, 2022.

[35] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *CRYPTO'86*, volume 263 of *LNCS*, pages 186–194. Springer, Heidelberg, Aug. 1987.

[36] A. Gabizon. On the security of the BCTV pinocchio zk-SNARK variant. Cryptology ePrint Archive, Report 2019/119, 2019. https://eprint.iacr.org/2019/119.

[37] A. Gabizon, Z. J. Williamson, and O. Ciobotaru. PLONK: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953, 2019. https://eprint.iacr.org/2019/953.

[38] An elastic proof system based on arkworks. https://github.com/arkworks-rs/gemini, 2022.

[39] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 626–645. Springer, Heidelberg, May 2013.

[40] C. Gentry, S. Halevi, and V. Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. In O. Dunkelman and S. Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 458–487. Springer, Heidelberg, May / June 2022.

[41] A. Ghoshal and S. Tessaro. Tight state-restoration soundness in the algebraic group model. In T. Malkin and C. Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 64–93, Virtual Event, Aug. 2021. Springer, Heidelberg.

[42] M. Girault. Self-certified public keys. In D. W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 490–497. Springer, Heidelberg, Apr. 1991.

[43] A. Golovnev, J. Lee, S. Setty, J. Thaler, and R. S. Wahby. Brakedown: Linear-time and post-quantum SNARKs for R1CS. Cryptology ePrint Archive, Report 2021/1043, 2021. https://eprint.iacr.org/2021/1043.

[44] Grin. https://github.com/mimblewimble/grin, 2022.

[45] J. Groth. On the size of pairing-based non-interactive arguments. In M. Fischlin and J.-S. Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 305–326. Springer, Heidelberg, May 2016.

[46] T. Haines, S. J. Lewis, O. Pereira, and V. Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy*, pages 644–660. IEEE Computer Society Press, May 2020.

[47] Harmony. The first go implementation of verifiable delay function (VDF). https://github.com/harmony-one/vdf, 2022.

[48] D. Hosszejni. Verifiable delay function as part of a master thesis. https://github.com/hdarjus/master-thesis-ELTE, 2022.

[49] Hyrax reference implementation. https://github.com/hyraxZK/hyraxZK, 2022.

[50] iden3. zksnark implementation in javascript & WASM. https://github.com/iden3/snarkjs, 2022.

[51] Incognito chain. https://incognito.org/, 2023.

[52] ING Bank. Reusable library for creating and verifying zero-knowledge range proofs and set membership proofs. https://web.archive.org/web/20201111215751/https://github.com/ing-bank/zkrp, 2022.

[53] IOHK. Sonic protocol. https://github.com/input-output-hk/sonic, 2022.

[54] IOTA Ledger. Implementation of verifiable delay function. https://github.com/iotaledger/vdf, 2022.
[55] T. E. Jedusor. Mimblewimble. https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt, 2016.
[56] A. Kate, G. M. Zaverucha, and I. Goldberg. Constant-size commitments to polynomials and their applications. In M. Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, Dec. 2010.
[57] A. Kothapalli, S. Setty, and I. Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In Y. Dodis and T. Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 359–388. Springer, Heidelberg, Aug. 2022.
[58] LayerX. Sonic implementation in rust. https://github.com/LayerXcom/lx-sonic, 2022.
[59] Litecoin. https://github.com/ltc-mweb/litecoin, 2022.
[60] C++ implementation of vss using lwe encryption and proofs. https://github.com/shaih/cpp-lwevss, 2021.
[61] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 2111–2128. ACM Press, Nov. 2019.
[62] Matter Labs. Bellman zksnark library for community with ethereum's bn256 support. https://github.com/matter-labs/bellman, 2022.
[63] Microsoft. Nova: Recursive snarks without trusted setup. https://github.com/microsoft/Nova, 2022.
[64] Microsoft. Spartan: High-speed zksnarks without trusted setup. https://github.com/microsoft/Spartan, 2022.
[65] J. Miller. Coordinated disclosure of vulnerabilities affecting girault, bulletproofs, and plonk. https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/, 2022.
[66] Mina protocol. https://minaprotocol.com/, 2022.
[67] Mir Protocol. Recursive snarks based on plonk and halo. https://github.com/mir-protocol/plonky, 2022.
[68] Monero: the secure, private, untraceable cryptocurrency. https://github.com/monero-project/monero, 2022.
[69] C. Network. Chia VDF utilities. https://github.com/Chia-Network/chiavdf, 2022.
[70] P. Network. An implementation of verifiable delay functions in rust. https://github.com/poanetwork/vdf, 2022.
[71] O(1) Labs. The proof systems used by mina. https://github.com/o1-labs/proof-systems, 2022.
[72] B. Parno. A note on the unsoundness of vnTinyRAM's SNARK. Cryptology ePrint Archive, Report 2015/437, 2015. https://eprint.iacr.org/2015/437.
[73] B. Parno, J. Howell, C. Gentry, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013.
[74] J. Pascoal and S. M. de Sousa. Bulletproofs-Ocaml. https://gitlab.com/releaselab/bulletproofs-ocaml/, 2022.
[75] A. Poelstra. https://github.com/apoelstra/secp256k1-zkp/tree/bulletproofs, 2022.
[76] D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, June 2000.
[77] Polygon. https://polygon.technology/, 2022.
[78] Python3 implementation of bulletproofs. https://github.com/wborgeaud/python-bulletproofs, 2022.
[79] C.-P. Schnorr. Efficient identification and signatures for smart cards. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, Aug. 1990.
[80] Scroll. https://scroll.io/, 2022.
[81] SECBIT Labs. Zero knowledge proofs toolkit for CKB. https://github.com/sec-bit/ckb-zkp, 2022.
[82] S. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 704–737. Springer, Heidelberg, Aug. 2020.
[83] O. Shlomovits. Javascript code for one-round single bulletproof. https://github.com/omershlo/simple-bulletproof-js, 2022.
[84] Starkware. https://starkware.co/, 2022.
[85] Tari bulletproofs+. https://github.com/tari-project/bulletproofs-plus, 2023.
[86] D. Unruh. Non-interactive zero-knowledge proofs in the quantum random oracle model. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 755–784. Springer, Heidelberg, Apr. 2015.
[87] W. Vasquez. Bulletproofs implementation in Go. https://github.com/wrv/bp-go, 2022.
[88] D. Wagner. A generalized birthday problem. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 288–303. Springer, Heidelberg, Aug. 2002.
[89] R. S. Wahby, I. Tzialla, a. shelat, J. Thaler, and M. Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.
[90] B. Wesolowski. Efficient verifiable delay functions. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019.
[91] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 733–764. Springer, Heidelberg, Aug. 2019.
[92] Dusk network. https://dusk.network/, 2022.
[93] ZCash. halo2. https://github.com/zcash/halo2, 2022.
[94] Zcash. https://z.cash/, 2022.
[95] ZenGo-X. A collection of paillier cryptosystem zero knowledge proofs. https://github.com/ZenGo-X/zk-paillier, 2022.
[96] Zengo x bulletproofs. https://github.com/ZenGo-X/bulletproofs, 2023.
[97] ZK Garage. A pure rust plonk implementation using arkworks as a backend. https://github.com/ZK-Garage/plonk, 2022.
[98] Implementation of bulletproof for zk spl token. https://github.com/DescartesNetwork/zkSen, 2023.