Fiat-Shamir in the Wild*

Hieu Nguyen[1,2], Uyen Ho[1], and Alex Biryukov[2]

[1] Verichains
[2] DCS and SnT, University of Luxembourg
alex.biryukov@uni.lu

Abstract. The Fiat-Shamir transformation is a key technique for removing interactivity from cryptographic proof systems in real-world applications. In this work, we discuss five types of Fiat-Shamir-related protocol design errors and illustrate them with concrete examples mainly taken from real-life applications. We discuss countermeasures for such vulnerabilities.

* For the purpose of open access and in fulfillment of the obligations arising from the grant agreements, the authors have applied a Creative Commons Attribution 4.0 International (CC BY 4.0) license to any author-accepted manuscript version arising from this submission.

1 Introduction

Fiat-Shamir transformation [8] is a well-known technique to remove interactivity from interactive public-coin proof systems. In these systems, the verifier generates and sends to the prover random values acting as challenges which can only be solved if the statement being proved is correct (the soundness property). If these random values can be predicted, it is likely that the prover will be able to forge proofs for incorrect statements.

The way the Fiat-Shamir transformation works is that it replaces random values with outputs from a cryptographic hash function[3], hashing the verifier context whenever randomness is needed. Public parameters, the statement being proved, previously exchanged messages, etc. all contribute to this context. Since randomness is removed, the verifier becomes deterministic and redundant, thus making the proof system non-interactive.

The convenience brought by the Fiat-Shamir transformation does not come without a price. It is fair to say that some security has been sacrificed because of a larger attack surface when applying the Fiat-Shamir transformation. Not only must the hash function in use be collision-resistant, the data fed into it also needs to be correct and complete.

Recall that in probabilistic proof systems, knowing the verifier challenges in advance helps the prover to forge proofs. As a result, vulnerabilities in Fiat-Shamir implementations usually cause the security of these systems to be completely broken.

Another important thing to note is that the Fiat-Shamir transformation allows for brute-force attacks. A malicious prover does not pay for a failed attempt to forge a proof. Thus, if a design or implementation flaw increases the proof system's soundness error from a negligible value to, for example, 2^{-32}, forging a proof can become very practical.

In Figure 1, a classification of the vulnerabilities reported in this work is given. The classification is based on the additional attack surfaces introduced by applying the Fiat-Shamir transformation. Surprisingly, most vulnerabilities have multiple case studies found in prominent open-source zero-knowledge proof or multi-party computation projects.

In the following sections, we will describe in detail different types of practical Fiat-Shamir transformation-related pitfalls. More precisely, we will identify the root cause, provide a method to forge proof under a vulnerable cryptographic scheme, mention its real-world impact, and provide suggestions for countermeasures. A quick summary is given in Table 1.

1.1 Comparison to related works

This work can be seen as an extension to both [7] and Section 5.3 of [17]. In [7], only the missing-public-statement vulnerability is extensively studied. Other

[3] Throughout this work, we assume that the Fiat-Shamir transformation requires a cryptographic hash function although this is not always the case [5].

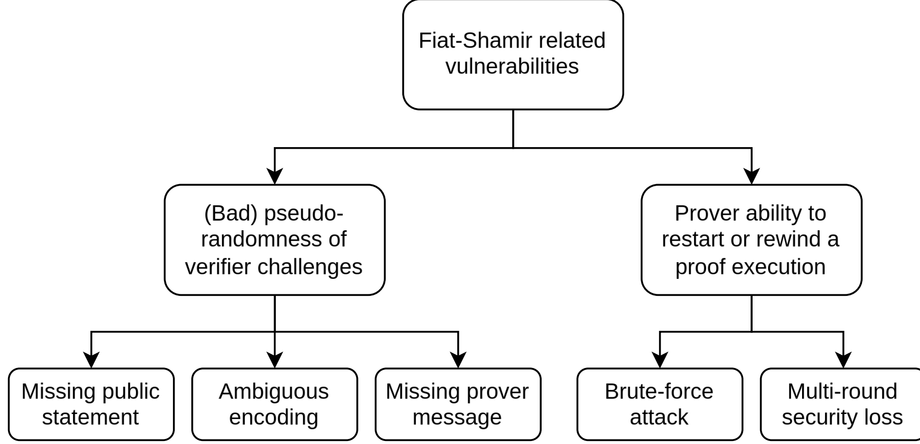

Fig. 1. A classification of the Fiat-Shamir-transformation-related vulnerabilities based on additional attack surfaces. Bad pseudo-randomness of verifier challenges usually comes from an incorrect implementation of the Fiat-Shamir transformation, and allows a direct attack. On the other hand, the unavoidable ability of the prover to restart or rewind a proof execution may allow for a practical attack to other protocol design and implementation weaknesses.

Table 1. The Fiat-Shamir-transformation-related vulnerabilities mentioned in this survey.

Vulnerability	Known practical exploit	Remedy
Missing public statement	Sec. 2, [2, 7, 12]	Public statement must be hashed.
Ambiguous encoding	Sec. 3, [15]	To-be-hashed data must be decodable.
Missing prover message	Sec. 4, [6]	All previous prover messasges must be hashed.
Brute-force attack	Sec. 5, [15]	Soundness error must be carefully considered.
Multi-round security loss	Sec. 6	Attacker can not win round-by-round.

vulnerabilities are only briefly mentioned (or not at all). We stress that the vulnerabilities covered in our paper are all Fiat-Shamir-transformation related, having the same severity (all lead to proof forging) and, therefore, should deserve equal attention.

In [17], several of the vulnerabilities are discussed from the theoretical point of view. By adding multiple concrete examples, this work acts as a practical complement to it.

2 Missing public statement

This type of vulnerability, usually known as the weak variant of the Fiat-Shamir transformation, can be found in many modern proof systems [2, 7, 12].

2.1 Root cause

Fiat-Shamir hashing context always includes the public statement to be proved. If this is not the case, a malicious prover will be able to change the statement without affecting verifier challenges. This will lead to broken protocol soundness if the prover can pick which statement to prove. In practice, this is typically the case. And if the prover can profit from proving a random[4] false statement, the consequences can be severe.

2.2 Case study: PLONK

The scheme PLONK [9] (Permutations over Lagrange-bases for Oecumenical Non-interactive arguments of Knowledge) is a SNARK (Succinct Non-interactive ARgument of Knowledge) released in 2019. Since then, PLONK has been widely adopted due to its extensibility to support beyond-arithmetic polynomial constraints such as permutation (polynomials' evaluations over a domain is the same as theirs or other polynomials' under an optionally predefined permutation) or lookup (each evaluation of a polynomial over a domain is contained in a table possibly made up from the evaluations of another polynomial).

This section provides only some of the main points on the technical side of PLONK that are required to understand the related attacks. Additional details will be provided on the fly when needed. For the full specification, see [9].

First of all, PLONK makes use of a polynomial commitment scheme that allows a prover to commit to a polynomial and, later on, prove to a verifier the correctness of the committed polynomial's evaluations at certain points chosen by the verifier.

Originally designed to prove the correctness of computation over an arithmetic circuit that supports only addition and multiplication gates, the (simplified) Fiat-Shamir-transformed PLONK prover algorithm is as follows. Recall that challenges

[4] The prover does not have full control over the statement for which he can forge a proof since it depends on verifier challenges derived from a hash function. As a result, the statement (or part of it) should look random.

are derived by invoking a cryptographic hash function with the current verifier context as input.

- Round 1: Compute 3 wire polynomials $a(X), b(X), c(X)$ and output their polynomial commitments. These polynomials are constructed from the values associated with the wires of the circuit, captured during an honest execution of the computation.
- Round 2: Derive permutation challenges β, γ, compute the permutation polynomial[5] $z(X)$ and output its polynomial commitment.
- Round 3: Derive a quotient challenge α, compute the quotient polynomial $t(X)$ (as described in Equation 1 below), and output its polynomial commitment.
- Round 4 & 5: Derive an evaluation challenge ξ, evaluate the committed polynomials at the desired points (ξ and others), and output the evaluations together with their proofs of correctness.

Finally, the PLONK verifier checks that:

- All polynomial evaluation proofs are correct.
- All polynomial constraints are satisfied when evaluated at ξ. In other words, the following aggregate constraint must hold at ξ:

$$
\begin{aligned}
t(X) \quad = & \frac{1}{Z_H(X)}(a(X)b(X)q_M(X) + a(X)q_L(X) + b(X)q_R(X) \\
& \quad + c(X)q_O(X) + q_C(X) + P(X)) \\
& + \frac{\alpha}{Z_H(X)}((a(X) + \beta X + \gamma)(b(X) + \beta k_1 X + \gamma) \\
& \quad (c(X) + \beta k_2 X + \gamma)z(X)) \\
& + \frac{-\alpha}{Z_H(X)}((a(X) + \beta S_{\sigma 1}(X) + \gamma)(b(X) + \beta S_{\sigma 2}(X) + \gamma) \\
& \quad (c(X) + \beta S_{\sigma 3}(X) + \gamma)z(X\omega)) \\
& + \frac{\alpha^2}{Z_H(X)}((z(X) - 1)L_1(X))
\end{aligned} \tag{1}
$$

where $P(X)$ is the polynomial constructed from the public input/output signals of the circuit. Other unmentioned terms are public or circuit-related parameters known to both the prover and the verifier. For example, the polynomials $q_M, q_L, q_R, q_O, q_C, S_{\sigma 1}, S_{\sigma 2}, S_{\sigma 3}$ are defined by the circuit.

The flaw The vulnerability is fairly simple: $P(X)$ is not included in the Fiat-Shamir hashing context. This means that changing $P(X)$ will not result in rerandomization of the challenges $(\beta, \gamma, \alpha, \xi)$.

[5] From protocol design perspective, a permutation constraint over the wire polynomials will be converted to arithmetic constraints involving this polynomial, at the cost of an additional round (this round).

The attack The attack strategy is as follows:

1. Pick arbitrary $a(X), b(X), c(X), z(X), t(X)$. All challenges are now determined.
2. Deduce a value p_ξ such that the aggregate constraint (1) is satisfied at ξ when $P(\xi) = p_\xi$.
3. Choose a suitable $P(X)$ such that $P(\xi) = p_\xi$.
4. Follow the rest of the prover algorithm (Round 4 & 5) as usual.

Note that Step 3 is achievable only if there is at least one public signal, say $P(1)$, that can be randomized (i.e. it can take an arbitrary value and can still pass further application-level logic checks with high probability) so that $P(\xi)$ and other public signals can be fixed to the desired values. Whether or not this condition can be satisfied is application-specific. In practice, it is actually very likely to hold. For example, a privacy-preserving payment application should have one of the public signals acting as a random nonce to prevent double spending transactions.

2.3 Real-world impact

Not only PLONK, but also many proof systems including Bulletproofs, Hyrax, Spartan, Wesolowski's VDF, etc. have implementations vulnerable to this flaw; and it is even possible to create unlimited money on a blockchain network by exploiting the weakness [7]. Older victims include voting protocols such as sVote [12] and Helios [2].

2.4 Remedy

The public statement that is being proved must be included in the Fiat-Shamir hashing context. Public parameters should also be considered as part of the public statement. If the statement is too large (e.g., a statement about the honest computation of a large circuit), hashing its succinct form is fine. One needs to ensure that the changeable parts of a statement are all involved in the generation of the challenges. Note that statements are usually made from filling in an application-specific template, and changeable parts are those that appear at the placeholders.

3 Ambiguous encoding

The case study given for this vulnerability was discovered by a team led by the first author. It was presented at Blackhat USA 2023 as part of TSSHOCK [15].

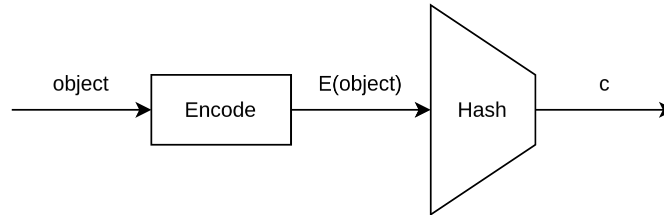

Fig. 2. Applying the Fiat-Shamir transformation involves an encoding scheme.

3.1 Root cause

Deriving a random challenge with the Fiat-Shamir transformation always involves some encoding scheme applied to an `object`, usually a list of integers. The encoding result `E(object)` is then fed into a cryptographic hash function `H` (see Figure 2).

In case the encoding is ambiguous, that is, different `object`s may have the same `E(object)`, the whole transformation process will never be collision resistant: one can choose `object1` and `object2` such that `H(E(object1))` is equal to `H(E(object2))` regardless of `H`.

3.2 Case study: dlnproof

The scheme *dlnproof* (formalized as the Ring-Pedersen Parameters proof in [4]) is an argument system used for proving knowledge of a discrete log modulo a composite number N. The protocol is similar to Schnorr's [16] and is given in Figure 3. There are 2 main differences:

- The group order $\phi(N)$ is unknown to the verifier.
- The challenge space is only $\{0, 1\}$.[6] As a result, proof repetition is required to achieve negligible soundness error. This is done in parallel: the prover sends $\alpha_0, \alpha_1, ...$, the verifier replies with $c_0, c_1, \ldots$, and the prover sends $\tau_0, \tau_1, \ldots$. The verifier only accepts if all triples (α_i, c_i, τ_i) are valid. In this case, the Fiat-Shamir hashing context consists of $N, g, h, \alpha_0, \alpha_1, \ldots$.

The flaw The following Golang code snippet is taken from a vulnerable library:

[6] This can be seen as a consequence of having the group order unknown to the verifier. In Schnorr's protocol, there is an important security argument called extractibility: If the prover can solve two different challenges c_1, c_2 under the same α, one can extract the secret discrete log value from the prover's messages (hence, the prover must know the secret). However, this only works when the difference between the two challenges $\Delta c = c_1 - c_2$ is invertible modulo the group order. When the group order is unknown, Δc must be ± 1 to ensure this property.

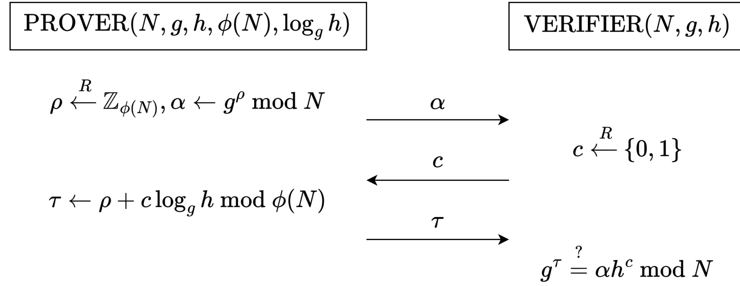

Fig. 3. Schnorr-like protocol for proving knowledge of $\log_g h$ modulo N.

```
for i := range in {
    data = append(data, ptrs[i]...)
    data = append(data, hashInputDelimiter) // safety delimiter
}
```

Here, `data` holds the result of encoding a list containing $N, g, h, \alpha_0, \alpha_1, \ldots$. It is constructed by joining `ptrs[i]` with a constant delimiter. In fact, each `ptrs[i]` is the byte representation corresponding to one of the integers. Clearly, this encoding scheme is ambiguous. For example, letting the delimiter be `"(DELIM)"`, then `["a(DELIM)a", "a"]` and `["a", "a(DELIM)a"]` are both encoded to the same byte string: `"a(DELIM)a(DELIM)a"`.

The attack The idea to forge a proof is as follows:

1. Prepare the payloads a_0, a_1 such that a_0, a_1 helps forge proof when $c = 0$ and $c = 1$ respectively, and their byte representations satisfy a condition depicted in Figure 4.
2. Only assign a_0 or a_1 to α_i.
3. If the number of challenge bits 0 (or 1) is equal to the number of a_0 (or a_1), rearrange the list of α_i according to the challenge bits to forge proof (see Figure 5). Otherwise, go back to step 1 and retry.

It can be seen that shuffling the list of α_i does not change its byte representation due to the vulnerability. Consequently, the challenges c_i also do not change.

3.3 Real-world impact

The interesting thing about dlnproof is that it is a building block of the well-known GG threshold ECDSA design [4,10,11] which is followed by many threshold ECDSA libraries in practice, and the ambiguous encoding flaw eventually breaks the security of all affected libraries: A malicious party can recover the private key of a TSS (threshold signature scheme) group using only a single signature [15].

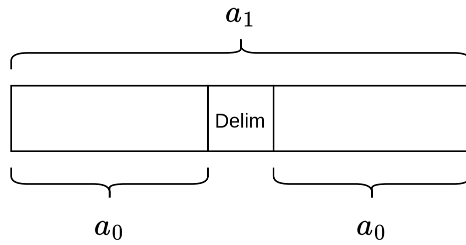

Fig. 4. Byte representations of a_0 and a_1.

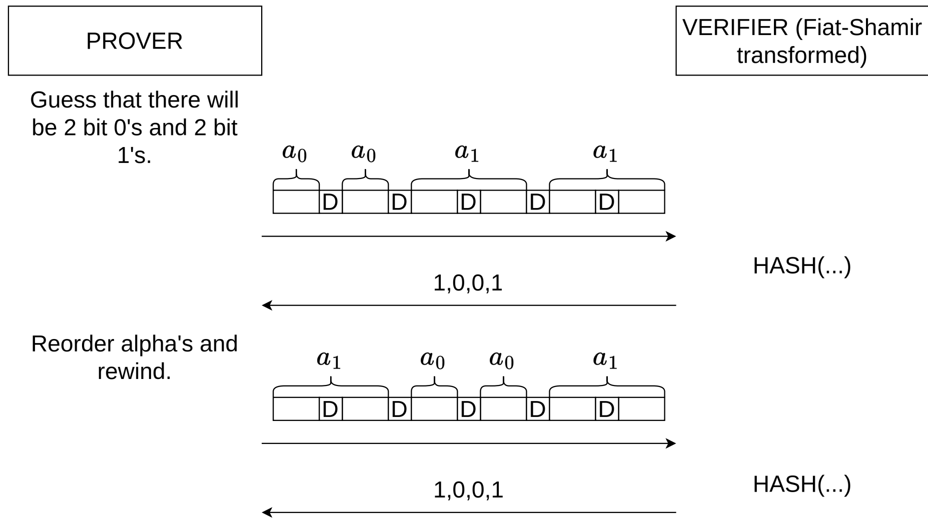

Fig. 5. An example of alpha-shuffling in the case #repetitions = 4.

As of mid-2023, many threshold ECDSA libraries, including those maintained by Binance, Taurus, Thorchain, etc., were found vulnerable to this flaw. Note that threshold signature is an important technology, largely used in the blockchain ecosystem to share the ownership of large funds. Potentially millions to billions of USD worth of cryptocurrency were found to be at risk. For more details, see [18].

3.4 Remedy

Always use a proper (unambiguous) encoding scheme combined with a collision-resistant hash function when performing the Fiat-Shamir transformation. A rule of thumb is to make sure that the to-be-hashed data is uniquely decodable. Broken hash functions (e.g. MD5, SHA-1), for which collisions have been found, must also be avoided.

4 Missing prover messages

As the case study given for this vulnerability is published for the first time, we provide a slightly more complete report compared to the others. For example, its vulnerable code is included (see Appendix).

4.1 Root cause

In a multi-round protocol, it is important that all the messages sent by the prover in the previous rounds are included in the hashing context. Missing any of them may lead to proof forging, as a malicious prover can rewind and change that message without affecting the currently generated challenge.

4.2 Case study: PLONK

The scheme The basics of PLONK are already described in Section 2.2.

The flaw Here is a short summary of how challenges are computed from a vulnerable library (for the original code snippet, see Appendix):

- $\beta = H(\text{public inputs, commitments of } a(X), b(X), c(X))$
- $\gamma = H(\beta)$
- $\alpha = H(\text{commitment of } z(X))$
- $\xi = H(\text{commitment of } t(X))$
- $v = H(\text{polynomial evaluations})$
- $u = H(\text{polynomial evaluation proofs})$

where H denotes a cryptographic hash function. As we can see, only the last prover message is given as input to the hash function when generating a challenge. Certainly, this Fiat-Shamir transformation is insecure.

The attack The goal is to make (1) satisfied when $X = \xi$ for arbitrary $P(X)$. Because of the flawed Fiat-Shamir transformation, it is possible to do so:

1. Pick an arbitrary $t(X)$ and compute the evaluation challenge ξ which depends only on $t(X)$.
2. Choose $z(X)$ such that it is evaluated to 0 at ξ and $\xi\omega$. This will eliminate β, γ from (1). Now α, which only depends on $z(X)$, is also determined.
3. Finally, pick suitable $a(X), b(X), c(X)$ such that $a(\xi), b(\xi), c(\xi)$ satisfy (1). The rest of the prover algorithm (rounds 4 & 5) can then be processed normally.

4.3 Real-world Impact

One of the most popular frameworks for working with SNARKs, SnarkJS, was found to be vulnerable to this flaw. At the time of writing, it was used by nearly 5400 projects on Github [14]. The vulnerability existed since May 2021 (the first version supporting PLONK, 0.4.0, was released) until May 2023 (a fix in version 0.7.0 was released). Currently, it is not known whether there is a real-world loss due to this mistake.

In another practical case study reported in [6], the last challenge is independent of the last prover message, leading to completely broken security of affected polynomial commitment schemes.

4.4 Remedy

Always ensure that all prover messages from previous rounds contribute to the Fiat-Shamir transformation process. One possible approach is to feed the last generated challenge (together with the most recent prover messages since then) into the hash function. This makes the output of all previous rounds recursively involved in the computation of the current challenge, thus eliminating the vulnerability.

5 Brute-force attack

Similarly to Section 3, the case study given for this vulnerability is also part of TSSHOCK [15].

5.1 Root cause

The soundness property of an interactive proof system means that a dishonest prover cannot convince the verifier if the statement being proved is false except with some small probability. It is often possible to amplify the soundness until the soundness error becomes negligible. This can be achieved by repeating the proof and accepting only if all proofs are valid.

When applying the Fiat-Shamir transformation to an interactive proof, the non-interactive version also inherits the soundness error. However, non-interactive proof allows the prover to generate the challenge deterministically by applying the hash function to a context. In this case, the malicious actor can just try different contexts until he/she finds a challenge that helps forging proof for the false statement without being detected by the verifier.

5.2 Case study: dlnproof with large challenge space

In this case study, a mistake in protocol design is presented, and the Fiat-Shamir transformation makes exploiting the mistake feasible.

Consider the interactive proof protocol dlnproof described in Section 2.2. dlnproof requires repeatedly generating a random challenge bit $c \in \{0, 1\}$. However, instead of repeating the interactive proof with binary challenge $c_i \in \{0, 1\}$, the vulnerable design decides to use a much larger set of challenges in only a single run (e.g., c is sampled from $\mathbb{Z}_{2^{256}}$ instead of $\mathbb{Z}_2$).

The soundness error in this protocol is expected to be inversely proportional to the size of the challenge space. Unfortunately, it turns out that a larger challenge set does not result in a better soundness error.

As an explanation, suppose that $g, h \in \mathbb{Z}_N^*$, $g = h^2$, and $2 \mid \text{ord}(g)$. Since 2 has no inverse modulo $\text{ord}(g)$, $\log_g h = \frac{1}{2}$ does not exist. However, a proof for the existence of $\log_g h$ can still be forged whenever $2 \mid c$ by having $\tau = \rho + \frac{c}{2}$. Therefore, the soundness error is still $\frac{1}{2}$ in this case.

Since repeating the original dlnproof 128 times seems slow, many popular threshold ECDSA libraries following the GG design [4, 10, 11] use this weak version of dlnproof to save some computation. The point is that exploiting this weakness in practice will not be easy. If one wants to recover the TSS private key with only a few signatures, the malicious exponent (equals 2 in the above example) needs to be large[7] (e.g., 2^{32}). This results in a lower soundness error ($\frac{1}{2^{32}}$) and, more importantly, once a proof forging attempt fails, the attacker will be immediately punished.

However, when the Fiat-Shamir transformation is applied, the attacker could simply keep restarting the weak dlnproof version until proof forging succeeds. This eventually breaks the security of all affected threshold ECDSA libraries [15].

5.3 Real-world impact

Threshold ECDSA libraries by Zengo, Axelar, ING Bank, ... were found to run into this pitfall [15]. The impact is similar to that of Section 3.3, except that more signatures are needed to recover the private key of a TSS group.

[7] In more details, during the signing protocol of the GG design, the attacker can learn $x \bmod e$ where x is a a known multiple of the ECDSA nonce and e is the malicious exponent. As a result, the ECDSA nonce leakage attack [13] can be applied. It is clear that the larger the exponent, the more information is leaked and thus the less signatures are required to launch the attack.

5.4 Remedy

The Fiat-Shamir transformation applied to an interactive protocol creates an attack surface for a malicious prover to mount a brute-force attack, thus the soundness error of the protocol must be carefully chosen and enforced.

6 Multi-round security loss

6.1 Root cause

If the Fiat-Shamir transformation is applied to a multi-round protocol, a malicious prover will be able to rewind the proving process to any previous round, keeping every generated challenge before that round fixed (called a state restoration attack in [1]). This is opposed to the interactive context in which the prover can only start at the beginning and all challenges have to be freshly generated.

As a consequence, the security of an interactive, multi-round protocol could be tremendously reduced (negligible soundness error becomes non-negligible) when the Fiat-Shamir transformation is applied.

6.2 Case study: Sequential repetition of dlnproof

Consider the case in which the dlnproof protocol is repeated sequentially λ times to achieve a $2^{-\lambda}$ soundness error. It is important to note that this construction is secure under the interactive setting. However, when the Fiat-Shamir transformation is applied, a malicious prover can carry out the brute force attack round-by-round. As a result, only a polynomial amount of work ($O(\lambda)$, on average) is required to forge a proof.

6.3 Remedy

It is always important to check if applying the Fiat-Shamir transformation is appropriate. From a practical point of view, the two graphs of prover states presented in Figure 6 give some hints to this. Let P denote the dishonest prover. Let $A \to B$ denote the event that P moves from state A to B. $\Pr[S_0 \to \texttt{acc}]$, the so-called soundness error, must be negligible in both cases.

In Protocol 2, we have $\Pr[S_0 \to \texttt{acc}] < \Pr[S_1 \to \texttt{acc}]$ since S_1 is closer to *accept*. Therefore, it makes sense for P, after the transitions $S_0 \to S_1 \to \texttt{rej}$, to rewind to S_1 instead of S_0 as less work needs to be undone and the probability of acceptance in S_1 is higher. As a result, the ability to rewind to any previous round does give P more power. The Fiat-Shamir transformation should be avoided in this case. Note that the sequential repetition of dlnproof is similar to Protocol 2 (the occurrence of a "good" challenge bit (probability $\frac{1}{2}$ per round) immediately causes P to fall into the *reject* state).

On the other hand, a similar analysis of Protocol 1 shows that there is no clear advantage in rewinding to S_1, so it might be possible to apply the Fiat-Shamir transformation in this case. In fact, Protocol 1 is similar to the round-by-round

soundness concept in [3] where the undecided states act as doomed states and all $\Pr[B_i]$ must be small to keep $\Pr[S_0 \rightarrow \texttt{acc}]$ negligible. Note that the PLONK protocol has a pattern similar to Protocol 1 (in each round of PLONK, a "bad" challenge can only occur with negligible probability but allows P to forge a valid proof, thus reaching the *accept* state).

When applying the Fiat-Shamir transformation is not appropriate, one can try switching to an alternative protocol (e.g., parallel instead of sequential repetition of dlnproof).

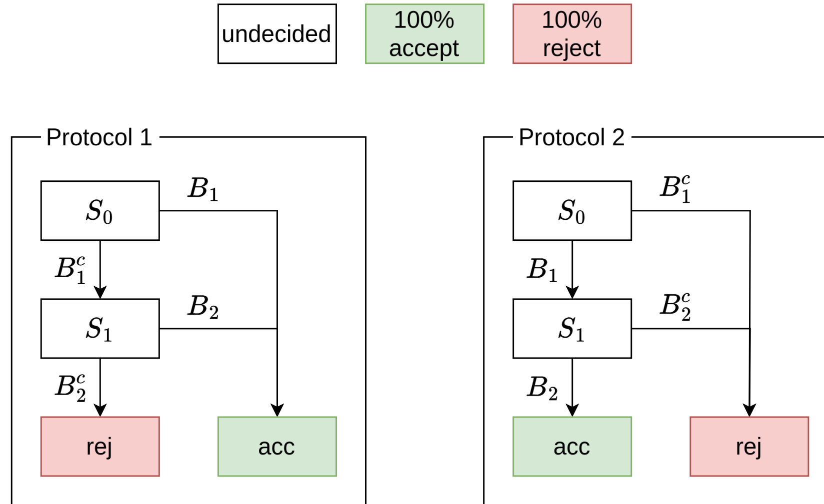

Fig. 6. Examples of protocol patterns that may be safe (Protocol 1) or unsafe (Protocol 2) to apply the Fiat-Shamir transformation. The graphs represent the state transitions of a dishonest prover starting in an undecided state S_0 due to imperfect soundness. B_i denotes the event that the verifier picks a "bad" challenge at round i, which increases the dishonest prover's chance to reach the *accept* state. In Protocol 1, B_i immediately leads to *accept* while in Protocol 2, B_i^c immediately leads to *reject*.

7 Responsible disclosure

The practical case studies of three out of five vulnerabilities presented in this survey were discovered by us recently.

- *Ambiguous encoding* and *brute-force attack*: Together, these two vulnerabilities affect many popular open-source threshold ECDSA libraries. Detailed information on the disclosure process can be found at [18].
- *Missing prover message*: We reported the vulnerability to team Iden3 (SnarkJS maintainer) in mid-2023. After confirming the exploitability, the team asked

us to delay the public disclosure to search for vulnerable projects. A fix for this vulnerability was planned prior to our report.

8 Acknowledgements

We would like to thank the anonymous reviewers at CSCML 2024 for their useful comments that help improve the paper. This research was funded in part by the Luxembourg National Research Fund (FNR), project CryptoFin C22/IS/17415825 it is in the scope of grant reference NCER22/IS/16570468/NCER-FT.

Appendix

This appendix contains the vulnerable code that demonstrates the missing prover message vulnerability described in Section 4.2. The code is written in Javascript and is extracted from the SnarkJS project at Github:

- Version: 0.6.11
- Path: `/src/plonk_verify.js`
- Function: `calculateChallanges`

```
const transcript1 = new Uint8Array(zkey.nPublic*n8r + G1.F.n8*2*3);
for (let i=0; i<zkey.nPublic; i++) {
    Fr.toRprBE(transcript1, i*n8r, A.slice((i)*n8r, (i+1)*n8r));
}
G1.toRprUncompressed(transcript1, zkey.nPublic*n8r + 0, proof.A);
G1.toRprUncompressed(transcript1, zkey.nPublic*n8r + G1.F.n8*2, proof.B);
G1.toRprUncompressed(transcript1, zkey.nPublic*n8r + G1.F.n8*4, proof.C);

ch.beta = hashToFr(transcript1);

const transcript2 = new Uint8Array(n8r);
Fr.toRprBE(transcript2, 0, ch.beta);
ch.gamma = hashToFr(transcript2);
...

const transcript3 = new Uint8Array(G1.F.n8*2);
G1.toRprUncompressed(transcript3, 0, proof.Z);
ch.alpha = hashToFr(transcript3);
...

const transcript4 = new Uint8Array(G1.F.n8*2*3);
G1.toRprUncompressed(transcript4, 0, proof.T1);
G1.toRprUncompressed(transcript4, G1.F.n8*2, proof.T2);
G1.toRprUncompressed(transcript4, G1.F.n8*4, proof.T3);
ch.xi = hashToFr(transcript4);
```

```
...

const transcript5 = new Uint8Array(n8r*7);
Fr.toRprBE(transcript5, 0, proof.eval_a);
Fr.toRprBE(transcript5, n8r, proof.eval_b);
Fr.toRprBE(transcript5, n8r*2, proof.eval_c);
Fr.toRprBE(transcript5, n8r*3, proof.eval_s1);
Fr.toRprBE(transcript5, n8r*4, proof.eval_s2);
Fr.toRprBE(transcript5, n8r*5, proof.eval_zw);
Fr.toRprBE(transcript5, n8r*6, proof.eval_r);
ch.v = [];
ch.v[1] = hashToFr(transcript5);
...

const transcript6 = new Uint8Array(G1.F.n8*2*2);
G1.toRprUncompressed(transcript6, 0, proof.Wxi);
G1.toRprUncompressed(transcript6, G1.F.n8*2, proof.Wxiw);
res.u = hashToFr(curve, transcript6);
...
```

References

1. Ben-Sasson, E., Chiesa, A., Spooner, N.: Interactive oracle proofs. In: Theory of Cryptography: 14th International Conference, TCC 2016-B, Beijing, China, October 31-November 3, 2016, Proceedings, Part II 14. pp. 31–60. Springer (2016)
2. Bernhard, D., Pereira, O., Warinschi, B.: How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In: Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18. pp. 626–643. Springer (2012)
3. Canetti, R., Chen, Y., Holmgren, J., Lombardi, A., Rothblum, G.N., Rothblum, R.D.: Fiat-Shamir from simpler assumptions. Cryptology ePrint Archive (2018)
4. Canetti, R., Gennaro, R., Goldfeder, S., Makriyannis, N., Peled, U.: UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1769–1787 (2020)
5. Chen, Y., Lombardi, A., Ma, F., Quach, W.: Does Fiat-Shamir require a cryptographic hash function? In: Annual International Cryptology Conference. pp. 334–363. Springer (2021)
6. Ciobotaru, O., Peter, M., Velichkov, V.: The last challenge attack: Exploiting a vulnerable implementation of the Fiat-Shamir transform in a KZG-based SNARK. Cryptology ePrint Archive (2024)
7. Dao, Q., Miller, J., Wright, O., Grubbs, P.: Weak Fiat-Shamir attacks on modern proof systems. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 199–216. IEEE (2023)
8. Fiat, A., Shamir, A.: How to prove yourself: Practical solutions to identification and signature problems. In: Conference on the theory and application of cryptographic techniques. pp. 186–194. Springer (1986)

9. Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive (2019)
10. Gennaro, R., Goldfeder, S.: Fast multiparty threshold ECDSA with fast trustless setup. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1179–1194 (2018)
11. Gennaro, R., Goldfeder, S.: One round threshold ECDSA with identifiable abort. Cryptology ePrint Archive (2020)
12. Haines, T., Lewis, S.J., Pereira, O., Teague, V.: How not to prove your election outcome. In: 2020 IEEE Symposium on Security and Privacy (SP). pp. 644–660. IEEE (2020)
13. Howgrave-Graham, N.A., Smart, N.P.: Lattice attacks on digital signature schemes. Designs, Codes and Cryptography **23**, 283–290 (2001)
14. Iden3: Snarkjs repository, `https://github.com/iden3/snarkjs/`
15. Nguyen, H., Nguyen, K., Nguyen, G., Nguyen, T., Nguyen, Q.: New key extraction attacks on threshold ECDSA implementations (2023), `https://github.com/verichains/tsshock/blob/main/verichains-tsshock-wp-v1.0.pdf`
16. Schnorr, C.P.: Efficient signature generation by smart cards. Journal of cryptology **4**, 161–174 (1991)
17. Thaler, J.: Proofs, arguments, and zero-knowledge. Foundations and Trends® in Privacy and Security **4**(2–4), 117–660 (2022)
18. Verichains: TSSHOCK, `https://www.verichains.io/tsshock/`