Notes on Proving the Security of Single-Party Schnorr

Chelsea Komlo

May 4, 2022

1 Introduction

This is an extremely informal review of Schnorr signatures and several methods to prove their security with respect to the discrete logarithm problem and variants thereof.

As a primer, Schnorr signatures are a fundamental cryptographic primitive in use today. Schnorr signatures are a sigma proof of knowledge of discrete log, made non-interactive and bound to some message m via the Fiat-Shamir transform. Sigma proofs of knowledge are characterised by a three-move protocol, where a prover begins by issuing a commitment, receives a challenge, and then issues a response. The proof is verified with respect to the commitment, challenge, and response. Notably, sigma proofs of knowledge can be made to be *zero-knowledge*, in that they allow the verifier to check the integrity of a proof with respect to some public information (the proof statement), without revealing private information (the proof witness) to the verifier.

2 Preliminaries

Let $\mathbb{G}$ be a prime-order group of order q and g be a generator for $\mathbb{G}$. $x \leftarrow y$ denotes the assignment of the value of y to x, and $x \xleftarrow{\$} S$ denotes sampling an element from the set S uniformly at random. $x \xleftarrow{\$} A()$ is the random variable x that is the output of a randomized algorithm A.

A Tiny Primer on Proving Security of Cryptographic Schemes When proving the security of cryptographic schemes, we want to demonstrate that the security of a particular scheme reduces to a hard mathematical problem. In the setting of schemes which are not information theoretically secure, security is modeled with respect to a computationally-bound adversary, which we assume is capable of performing polynomial-time operations efficiently, and that is probabilistic (i.e, the adversary has access to a random tape). We model the adversary as a Turing machine which can query oracles, keep its own internal state, and output some values at the end of its execution, with the goal of winning some pre-defined game.

In this note, we review reductions from the hardness of such an adversary producing a forgery of a Schnorr signature with respect to a given public key, to the hardness of solving for the discrete logarithm of an element within a prime-order group.

Random Oracle Model. The random oracle model (ROM) assumes that cryptographic hash functions (which we denote as H) operate as random oracles. In short, with each query $H(\alpha) \to \gamma_1, H(\beta) \to \gamma_2$, the outputs γ_1, γ_2 are assumed to be completely uncorrelated and act as values chosen at random from some (large enough to be hard to permute) set. In reality, we know that cryptographic hash functions are not so perfect, but this assumption is often critical to be able to demonstrate the security of many cryptographic primitives.

Forking Lemma. The forking lemma (introduced by Pointcheval and Stern) gives a lower bound to the probability of success when a probabilistic (allowed access to randomness) polynomial-time adversary is *rewound* to a specific point in a security game. At a high level, rewinding an adversary is a technique used when proving security to later extract sufficient information to demonstrate a reduction from the cryptographic scheme to a hard mathematical problem. Importantly, rewinding allows for "forking" an adversary's state, where it is guaranteed that the adversary outputs or operates over deterministic elements before the fork, whereas the adversary might diverge after the fork due to the introduction of fresh randomness. The forking lemma gives a hard bound on the probability that the adversary will succeed in a second execution, given the adversary's success in the first execution and access to the same random tape (i.e, the same source of randomness between the two executions).

More specifically, given a three-move sigma protocol Σ for which an adversary $\mathcal{A}$ has likelihood ϵ chance of success outputting a forgery, the forking lemma says that there exists a $1/\epsilon$ chance that for two runs of the protocol, the adversary will output the same first move (the commitment) but where the verifier outputs a distinct challenge between the two executions.

3 Schnorr Identification and Signature Schemes

First, we'll take a look at a Schnorr identification scheme, and then move on to Schnorr signatures.

3.1 Schnorr Identification Protocol

In this scheme, the prover simply proves knowledge of their secret key sk corresponding to the publicly known public key PK. The challenge is chosen dynamically by a separate entity which we refer to as the verifier.

The Schnorr identification scheme is as follows.

- *KeyGen*$(\lambda) \rightarrow (sk, PK)$: Sample the secret key $sk \xleftarrow{\$} \mathbb{Z}_q$ with respect to the security parameter λ; generate the public key $PK \leftarrow g^{sk}$. Output (sk, PK).
- *Prove*$(PK) \rightarrow \{0, 1\}$: Occurs as a three-move interactive protocol between the prover and the verifier.
    1. The prover begins by sampling a nonce $r \xleftarrow{\$} \mathbb{Z}_q$ and commitment $R \leftarrow g^r$, and sends R to the verifier.
    2. The verifier samples a challenge $c \xleftarrow{\$} \mathbb{Z}_q$ and sends c to the prover.
    3. The prover generates the response
    $$z \leftarrow r + c \cdot sk$$
    The prover then sends the response z to the challenger
    4. The verifier then checks that
    $$g^z \stackrel{?}{=} R \cdot PK^c$$
    If the check holds, the verifier outputs success, otherwise outputs failure.

The prover uses the randomizer r to ensure that their response does not allow the verifier to learn sk; i.e, to ensure that the protocol is zero-knowledge. Observe that the verifier learns nothing about sk other than that the prover knows the discrete logarithm of PK. Further, the protocol is a proof of knowledge, in that the prover cannot lie about their knowledge of sk.

3.2 Schnorr Signatures

To move to a signature of knowledge with respect to a message m, the Fiat-Shamir transform is used, to allow the signing operation to instead be *non-interactive*. In short, the Fiat-Shamir transform allows the prover to derive the challenge non-interactively via a hash function, instead of dynamically interacting with a challenger.

The Schnorr signature scheme is as follows.

- *KeyGen*$(\lambda) \rightarrow (sk, PK)$: Sample the secret key $sk \xleftarrow{\$} \mathbb{Z}_q$; generate the public key $PK \leftarrow g^{sk}$. Output (sk, PK).
- *Sign*$(PK) \rightarrow \sigma$: Derive the challenge as $c \leftarrow H(R, m, PK)$ and the response as $z \leftarrow r + c \cdot sk$. Output the signature $\sigma = (R, z)$.
- *Verify*$(PK, m, \sigma) \rightarrow \{0, 1\}$: The verifier derives the challenge as $c' \leftarrow H(R, m, PK)$, and then checks if $g^z \stackrel{?}{=} R \cdot PK^{c'}$. If the check holds, the verifier outputs 1, otherwise 0.

```
Simulation of Schnorr Signature Scheme
Signer($PK$)                                                    Verifier($PK$)
                                   $\xleftarrow{m}$
$z, c \xleftarrow{\$} \mathbb{Z}_q$
$R \leftarrow g^z \cdot PK^{-c}$
  // Derive $R$ relative to $PK$
$H(R, m) \leftarrow c$
  // Program the random oracle $H$
  // on inputs $R, m$ to output $c$
                                   $\xrightarrow{\sigma = (R, z)}$
                                                                $c' \leftarrow H(R, m)$
                                                                $g^z \overset{?}{=} R \cdot PK^{-c'}$
                                   // Because $H$ has already
                                   // been programmed, $c' = c$
                                   // The check holds true,
                                   // due to the derivation of $R$
```

Figure 1: Simulation of the Schnorr signature scheme. The environment receives a challenge $PK \in \mathbb{G}$ for which it does not know the discrete log; it then simulates signing with respect to PK to the adversary, which plays the role of the verifier.

4 Proof for Schnorr Signatures via ROM

We'll start first by reviewing the proof of security for Schnorr's signature scheme. In this setting, the proof of security requires operating within the random oracle model (ROM) and builds upon the forking lemma.

Proving the security of Schnorr signatures with respect to the discrete logarithm problem and the ROM is as follows, and is summarized in Figure 3.2. First, we employ a simulator *SIM* which simulates all actions of a signer to a probabilistic (given access to a random tape) computationally bound (polynomial time) adversary. *SIM* will operate in such a way that the adversary will believe that it is interacting with a real signer. Doing so is critical to demonstrating that the scheme is secure, otherwise an adversary in a security experiment might act differently from a real adversary attacking the scheme. Here, instead of performing the Schnorr key generation directly as described in Section 3.2, *SIM* instead accepts a challenge $Y \in \mathbb{G}$ whose discrete logarithm is unknown. To extract the dlog of Y, *SIM* will use as a subroutine an adversary $\mathcal{A}$ whose goal is to produce a forgery of a Schnorr signature with respect to some public key PK. In other words, *SIM* sets $PK = Y$, simulates the Schnorr signing protocol to $\mathcal{A}$, and then uses the forgeries produced by $\mathcal{A}$ to extract the discrete logarithm of Y. This is where the ROM is needed, because in order to simulate correctly, *SIM* must *program* the random oracle so that the signature is correct, but the adversary cannot distinguish the programmed random oracle from a real execution.

In doing so, we can claim that producing a forgery of a Schnorr signature for a particular public key is as hard as finding the discrete logarithm of Y. It is important that we treat $\mathcal{A}$ as a *black box*; no assumptions about *how* $\mathcal{A}$ produces these forgeries are made, just that $\mathcal{A}$ is capable of doing so efficiently.

In this simulation, *SIM* simulates two oracles that the adversary is allowed to query with inputs of its choosing.

1. First, *SIM* simulates the signing oracle $\mathit{Sign}(PK, m) \rightarrow \sigma$. The adversary provides as input the public key PK for which the signature is valid with respect to, and the message m that the message will be produced for.

2. Second, *SIM* simulates the random oracle H. This is important, as *SIM* is allowed to see all inputs to H and choose its outputs.

The only restriction on *SIM* for how to implement these oracles is that it must do so in a way that is *indistinguishable* to $\mathcal{A}$ from a real signer. We now describe how *SIM* implements these oracles.

To simulate H with arbitrary input from the adversary, *SIM* maintains an internal table Q that maps inputs to H with their respective outputs. When it receives a query, it first checks to see if an entry exists in Q. Otherwise, it samples an output ℓ from $\mathbb{Z}_q$, programs Q to return ℓ with respect to this particular input in the future, and returns ℓ.

To simulate *Sign* for each query made by the adversary with inputs (PK, m), the prover selects the response and challenge at random $z, c \xleftarrow{\$} \mathbb{Z}_q$. The prover then derives the commitment $R \leftarrow g^z \cdot PK^{-c}$. Importantly, the prover then *programs* H on inputs (R, m, PK) to output c. In the future, when $\mathcal{A}$ queries the random oracle as $H(R, m)$ (which it can do independently of querying the signature oracle), the adversary will receive c. Note that because c is chosen uniformly at random, the simulation is perfect. Finally, *SIM* outputs $\sigma = (R, z)$ as the signature. The simulation is perfect, as $g^z = R \cdot PK^c$.

Critical to *SIM*'s success is that it can program $H(R, m)$ strictly *before* the adversary is able to query for its output.

Extracting the discrete logarithm of the challenge. An adversary that is successful will eventually produce a forgery $(m^*, \sigma^* = (R^*, z^*))$ that is valid with respect to Y. Note the definition of a forgery is that σ^* is a valid signature, but was not queried to *Sign*. However, this forgery alone is not enough to extract the discrete logarithm of Y. So, *SIM re-winds* the adversary to the beginning of the experiment, providing it with the same random tape. However, *SIM* programs $H(R^*, m^*, PK)$ to instead output c', where $c' \neq c$. So, when the adversary produces their second forgery $(m^*, \sigma^* = (R^*, z^{**}))$, the forking lemma says that the adversary will produce such an output with some non-negligible probability.

And so *SIM* will output x, where $g^x = Y$, by deriving:

$$\frac{z^* - z^{**}}{(c^* - c^{**})}$$

Recalling that the response has the form $z = r + c \cdot sk$ and that z^*, z^{**} are with respect to the same commitment R^*, it becomes clear that the above equality evaluates to $sk = x$. And so an adversary that can produce a forgery of a Schnorr signature can be used as a subroutine to extract discrete logarithm of an arbitrary challenge value $Y \in \mathbb{G}$.

5 Proof for Schnorr Identification Scheme

Notice that in the proof for Schnorr signatures, *SIM* has the ability to select the challenge and program the random oracle before the adversary is allowed to see the challenge. However, in the setting for the Schnorr identification protocol, the prover receives a challenge that is selected by the verifier. And so the proof strategy for Schnorr signatures does not immediately apply to the proof for the Schnorr identification protocol, because the verifier learns the challenge before the signer does.

Because of this restriction, the Schnorr identification protocol requires the assumption that the verifier acts honestly when selecting the challenge.

To reflect this assumption in the proof of security, the adversary is simply given the *transcript* of the interaction between the prover and the verifier, and allowed to check that the protocol was executed honestly.

The adversary is required to produce an accepting transcript with respect to Y; however, *SIM* plays the role of the verifier (and so supplies the challenge to the adversary). Similar to before, if the adversary can produce two accepting transcripts (after rewinding), then the discrete logarithm of Y can be obtained as before.

6 Proof for Schnorr Signatures via AGM

Now let's look at proving the security of Schnorr signatures in the Algebraic Group Model (AGM). The AGM was formalized by Fuchsbauer, Kiltz, and Loss in 2019 [2], as a model to allow for tighter security reductions than what the standard model allows for. Recall that the standard model assumes only that the adversary is computationally bound and probabilistic. In a nutshell, the AGM requires an adversary that outputs a group element $A \in \mathbb{G}$ to also output the vector $\vec{a} = \langle a_0, \ldots, a_n \rangle$ representing how A is the linear combination $A = g^{a_0} C_1^{a_1} \ldots$ of all group elements $g, C_0, C_1, \ldots \in \mathbb{G}$ that the adversary has seen throughout its execution.

Importantly for the provability of Schnorr signatures, the AGM allows for a *straight-line* reduction, meaning that the adversary does not need to be rewound while still allowing for a reduction to the hardness of discrete log. This fact means that proofs in the AGM are less cumbersome and are easier to analyze, as we don't need to think through all edge cases associated with rewinding.

Similarly to in the ROM, the simulator *SIM* accepts as input a challenge Y and uses this challenge $PK \leftarrow Y$ as the public key in a simulation of Schnorr signing to an adversary. *SIM* again simulates the random oracle H and signing oracle *Sign* to the adversary. However, in this setting, when $\mathcal{A}$ outputs its forgery $\sigma^* = (R^*, m^*)$, it also outputs $\vec{a}$.

At this point, *SIM* can extract the discrete logarithm of Y without rewinding the adversary to generate a second forgery. Let's look at a simple example where $R^* = g^{a_0} PK^{a_1}$ (which is possible, as at the beginning of its execution the adversary is given knowledge of the generator and the public key). Recalling that the equality holds $R = g^z PK^{-c}$ for a valid signature, combining terms gives the following equation:

$$g^{a_0} PK^{a_1} = R = g^z PK^{-c}$$

Re-arranging:

$$PK^{a_1+c} = g^{z-a_0}$$

$$PK = g^{\frac{z-a_0}{a_1+c}}$$

Because *SIM* knows z, c^*, a_0, a_1, *SIM* can simply derive $sk = \frac{z-a_0}{a_1+c}$ directly, thereby learning the discrete log to the challenge.

7 Proof for Concurrent Schnorr Identification via OMDL

Informally, the one more discrete log (OMDL) problem can be stated as follows. After t queries to a challenge oracle that outputs a challenge Y_i and $t-1$ queries to the discrete logarithm oracle that outputs x_i such that $Y_i = g^{x_i}$, the challenge is to output *one additional* discrete logarithm x_t where Y_t was *not* queried to the discrete logarithm oracle. Note this problem can also be expressed where ordering is not assumed (i.e, the discrete logarithm oracle can be queried on any $t-1$ challenge values, so long as the solver produces t solutions corresponding to the t challenges).

The OMDL problem is useful for proving variants of Schnorr signature schemes when it becomes difficult for the prover to correctly program the random oracle. Such situations can occur when the adversary has more influence over the inputs that are provided to the random oracle. For example, in threshold signature schemes or blind signatures, the adversary is allowed to participate in the signing protocol, and so has some amount of influence over the value of

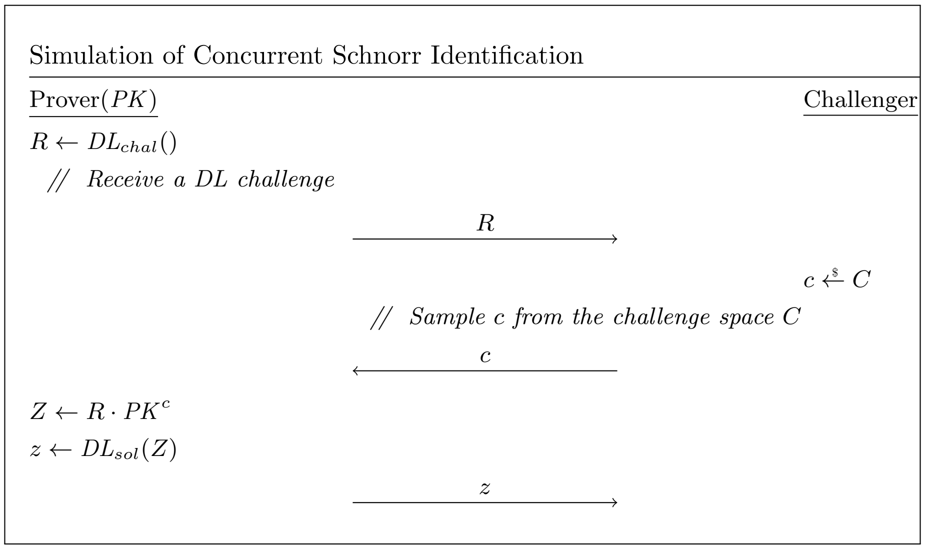
Simulation of Concurrent Schnorr Identification
Prover(PK)
Challenger
$R \leftarrow DL_{chal}()$
// Receive a DL challenge
R
$c \xleftarrow{\$} C$
// Sample c from the challenge space C
c
$Z \leftarrow R \cdot PK^c$
$z \leftarrow DL_{sol}(Z)$
z

Figure 2: Simulation of Schnorr Identification Protocol against Concurrent queries. The challenge space is denoted as C. The response z is derived from querying for the discrete logarithm solution to Z. The public key PK is similarly an output of DL_{chal}.

the challenge. In this setting, proving security becomes harder because the simulator does not know the value of the challenge c when it selects its commitment R. While OMDL allows for a useful proving mechanism in this setting, OMDL is a stronger assumption than plain dlog, which has tradeoffs when considering how realistic this assumption is in a practical setting.

The proof of Schnorr's identification scheme in a concurrent setting was given by Bellare and Palacio [1]. In this setting, the simulator is required to simulate the identification protocol as a series of some number of sessions, as opposed to performing a single execution. We denote the oracle that outputs a random discrete log challenge as $DL_{chal}() \rightarrow Y$ such that $Y \in \mathbb{G}$ and the oracle that outputs a discrete log solution as $DL_{sol}(Y) \rightarrow x$ such that $Y = g^x$.

The simulation of Schnorr's identification scheme in a concurrent setting is sketched in Figure 2, and is roughly as follows. First, *SIM* obtains PK via querying $PK \xleftarrow{\$} DL_{chal}()$. To issue a commitment R_i, *SIM* queries the oracle $R_i \xleftarrow{\$} DL_{chal}()$. It then sends the verifier R_i. Upon receiving the challenge c_i from the verifier, *SIM* queries $DL_{sol}(R_i \cdot PK^{c_i})$, receiving z_i in return. It then sends z_i to the verifier. The transcript of this interaction is then sent to the adversary.

Let's assume the prover is required to perform ℓ concurrent iterations of the *Prove* operation against a challenger. The prover will have queried $DL_{chal}()$ and DL_{sol} each exactly ℓ times. Then, after providing the adversary with this transcript across two iterations with the same source of randomness, the environment receives two forgeries $(R^*, c^*, z^*), (R^*, c^{**}, z^{**})$ by the forking lemma. Using the same strategy as in prior proofs, the environment can extract *sk* from these two forgeries. Then, using knowledge of *sk* and each (R_i, c_i, z_i), the environment can solve for $R_1, \ldots, R_\ell$. Hence, an adversary that can produce forgeries against the Schnorr identification protocol in a concurrent setting could be used as a subroutine to solve the OMDL problem.

8 Acknowledgments

Many thanks to Deirdre Connolly for her help in reviewing and giving helpful feedback on this note.

References

[1] M. Bellare and A. Palacio. GQ and schnorr identification schemes: Proofs of security against impersonation under active and concurrent attacks. In M. Yung, editor, *CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002*, volume 2442 of *LNCS*, pages 162–177. Springer, 2002.

[2] G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018*, volume 10992 of *LNCS*, pages 33–62. Springer, 2018.