A Note on Various Forking Lemmas

Chelsea Komlo

April 24, 2023

1 Introduction

First introduced by Pointcheval and Stern [3], the forking lemma is commonly used in proofs of security that require *rewinding* an adversary in order to demonstrate a reduction to breaking some known-to-be-hard mathematical problem. In this note, we review the original forking lemma and several variations thereof. Each variation is made in the effort to provide a tighter proof of security for the context that it is used, or better abstraction as to allow for more general applications.

The intuition for the forking lemma is as follows. We begin with an adversary that is modeled as a probabilistic Turing machine $\mathcal{A}$ that is initialized with a random tape and access to a hash function modeled as a random oracle. While the behavior of the adversary is generally not defined (making the adversary "black box," the adversary outputs some value that will either satisfy some pre-defined conditions (thus winning the security game), or not satisfy these conditions. Note this setup requires the assumption that a hash function can be simulated by a truly random function, which is known as the "random oracle model" [4].

If $\mathcal{A}$ completes its attack successfully, we can lower bound the probability that $\mathcal{A}$ again completes successfully in a second execution with the same random tape but *different* outputs from the random oracle. Determining this lower bound on the success probability of $\mathcal{A}$ across two executions is important for many proofs of security, as the two outputs are then employed to demonstrate the reduction (i.e, solve for the discrete logarithm of a challenge).

In this note, we begin by first reviewing the original forking lemma by Pointcheval and Stern. We then look at several subsequent variants, and how each variant allows for better abstraction (and hence applications beyond signature schemes), tighter bounds in the security proof, or tailoring to the context of a specific security proof.

2 Forking Lemma by Pointcheval and Stern

The first variant of the forking lemma was introduced by Pointcheval and Stern in their proof of security for Schnorr signatures [3]. The lemma is with respect to a digital signature scheme in the random oracle model. More specifically, the lemma assumes an adversary modeled as a probabilistic Turing machine given access to a random tape (source of randomness) and a hash function modeled as a (programmable) random oracle. Then, the lemma demonstrates that if the adversary outputs a valid forgery σ for random oracle output h in its first execution, then with non-negligible probability, the adversary will output a valid second forgery with respect to a different different random oracle output h' with non-negligible success. Here, non-negligible specifically means the probability of success must be greater than $1/f(n)$, where $f(n)$ is any polynomial function.

Lemma 2.1. Forking Lemma. *Let $\mathcal{A}$ be a probabilistic polynomial time Turing machine given only public data as input. If $\mathcal{A}$ can find with non-negligible probability a valid signature σ with respect to a message m and random oracle output h, then, with non-negligible probability, if $\mathcal{A}$ is executed a second time with the same source of randomness but a different random oracle, then $\mathcal{A}$ will output a second valid forgery σ' for the same message m but with respect to a random oracle output h', such that $h \neq h'$.*

Pointcheval and Stern later give a different variant of the forking lemma [4] with respect to the adversary's probability of success ϵ and time bound T.

2.1 Applications

The forking lemma allows for demonstrating a reduction to discrete log for Schnorr signatures. Recall that a Schnorr signature is the tuple $\sigma = (R, z)$, such that $R = g^r$ and $z = r + c \cdot x$, where x is the secret key and $Y = g^x$ is the public key.

In the random oracle model, the simulator receives Y as the challenge and can simulate signing with respect to Y by programming the random oracle. After the simulator runs $\mathcal{A}$ and receives two valid forgeries $\sigma = (R, z)$ and $\sigma' = (R, z')$ with respect to Y, the simulator can output x by deriving:

$$\frac{z - z'}{(c - c')}$$

where $c = H(R, m)$ in the first execution of the adversary, and $c' = H(R, m)$ in the second execution of the adversary, but critically, such that $c \neq c'$. The fact that $c \neq c'$ cannot be detected by the adversary though, as it is executed twice, without keeping state between each execution.

3 General Forking Lemma

Unlike the forking lemma by Pointcheval and Stern, the general forking lemma introduced by Bellare and Neven [2] abstracts away the details of signature schemes and random oracles. Instead, the lemma simply asserts on the probability of some output of an algorithm when run on two related inputs.

```
Algorithm $\mathsf{GenFork}(X)$
Pick coins $\rho$ for $\mathcal{A}$ at random.
$\{h_1, \dots, h_q\} \xleftarrow{\$} H$   // Sample $q$ elements from $H$
$\mathsf{Q} \leftarrow \{h_1, \dots, h_q\}$
$(i, \mathsf{aux}) \xleftarrow{\$} \mathcal{A}(X, \mathsf{Q}; \rho)$
return $\perp$ if $i = \perp$
$\{h'_i, \dots, h'_q\} \xleftarrow{\$} H$
$\mathsf{Q}' \leftarrow \{h_1, \dots, h_{i-1}\} \cup \{h'_i, \dots, h'_q\}$
  // $\mathsf{Q}$ and $\mathsf{Q}'$ differ at $(q - i + 1)$ points
$(i', \mathsf{aux}') \xleftarrow{\$} \mathcal{A}(X, \mathsf{Q}'; \rho)$
return $\perp$ if $i = \perp$
if $i = i'$
  // Check that the output is with respect to the same "forked" index
  if $h_i \neq h'_i$
    // Check that a hash collision has not occurred.
    return $(1, \mathsf{aux}, \mathsf{aux}')$
return $\perp$
```

Figure 1: General Forking Algorithm

Let $q \geq 1$ be an integer, and H be a set. Let $X \in \mathbb{G}$, such that $X \xleftarrow{\$} \mathsf{IG}$, where IG is the instance generator. As before, let $\mathcal{A}$ be a randomized algorithm, that outputs $i \in \{\perp\} \cup \{1, \dots, q\}$, and auxiliary output aux. We denote the general forking algorithm as $\mathsf{GenFork}$, and show it below.

Let $\mathsf{acc}(X)$ denotes the probability that $\mathcal{A}$ completes successfully. I.e, acc denotes that $\mathcal{A}$ outputs a value that is accepted in its first execution in $\mathsf{GenFork}$ when given the input X. We show this probability in Equation 1.

$$\mathsf{acc}(X) = \Pr\left[\, i \geq 1 \;:\; \{h_1, \dots, h_q\} \xleftarrow{\$} H^q \;;\; (i, \mathsf{aux}) \xleftarrow{\$} \mathcal{A}(X, \{h_1, \dots, h_1\}) \,\right] \quad (1)$$

Let $\mathsf{ForkAcc}$ denote the probability that $\mathsf{GenFork}$ returns successfully given some $X \xleftarrow{\$} \mathsf{IG}$, as in Equation 2:

$$\mathsf{ForkAcc} = \Pr\left[\; b = 1 \;:\; X \xleftarrow{\$} \mathsf{IG} \;;\; (b, \mathsf{aux}, \mathsf{aux}') \xleftarrow{\$} \mathsf{GenFork}(X) \;\right] \quad (2)$$

Bellare and Neven then demonstrate that Equations 1 generalize to any $X \xleftarrow{\$} \mathsf{IG}$. So Lemma 3.1 can be proved with respect to simply acc, as follows:

Lemma 3.1. *Given* acc *and* $\mathsf{ForkAcc}$*, as defined, then* $\mathsf{ForkAcc}$ *is lower bounded by* acc *as in Equation 3.*

$$\mathsf{ForkAcc} \geq \mathsf{acc} \cdot \left(\frac{\mathsf{acc}}{q} - \frac{1}{|H|} \right) \tag{3}$$

Alternatively:

$$\mathsf{acc} \leq \frac{q}{|H|} + \sqrt{q \cdot \mathsf{ForkAcc}} \tag{4}$$

3.1 Applications

While the general forking lemma models the adversary $\mathcal{A}$ as what is executed by the forking algorithm directly, when employed in a proof, there is need for an *intermediate* adversary which simulates the environment to the actual adversary which attacks the scheme. So in practice, $\mathsf{GenFork}(X)$ will execute some adversary $\mathcal{B}$, which will set up and simulate the environment (generally with respect to some challenge), such as a signing protocol. $\mathcal{B}$ will then execute $\mathcal{A}$ which actually attacks the scheme, interacting with $\mathcal{B}$ in order to do so.

Why the need for all of this indirection? We need to argue that $\mathcal{A}$ interacts with the scheme in a way that is *indistinguishable* from a real run of the protocol. The job of $\mathcal{B}$ is to ensure that, using the inputs of $\mathsf{GenFork}(X)$ to do so.

4 Local Forking Lemma

Unlike the generalized forking algorithm $\mathsf{GenFork}$ where Q and Q' differ by $q-i$ elements, the local forking lemma [1] is such that the sets employed as input to $\mathcal{A}$ differ by only a *single* element. Intuitively, this translates to the random oracle being re-programmed at only a single point after the fork, as opposed to every point after the fork. In particular, the oracle is re-programmed only at the index i output by $\mathcal{A}$ in its first execution. Why is this useful? The bounds in the proof of security can then be tighter, which allows for a more accurate representation of the security of the scheme.

We show the local forking algorithm in Figure 2:. The local forking algorithm gives (slightly) tighter bounds than the general forking lemma, as shown in Equation 5.

$$\mathsf{ForkAcc} \geq \frac{\mathsf{acc}^2}{q} \tag{5}$$

Lemma 4.1. *Given* acc *and* $\mathsf{ForkAcc}$*, as defined, then* $\mathsf{ForkAcc}$ *is lower bounded by* acc *as in Equation 5.*

5 Extension of Generalized Forking Lemma

In recent analysis of the FROST signature scheme, Bellare, Tessaro, and Zhu introduced an extension of the generalized forking lemma as well as an extension

```
Algorithm $\mathsf{LocalFork}(X)$
Pick coins $\rho$ for $\mathcal{A}$ at random.
$\{h_1, \ldots, h_q\} \xleftarrow{\$} H^q$
$\mathsf{Q} \leftarrow \{h_1, \ldots, h_q\}$
$(i, \mathsf{aux}) \xleftarrow{\$} \mathcal{A}(X, \mathsf{Q}; \rho)$
return $\perp$ if $i = \perp$
$h'_i \xleftarrow{\$} H$
$\mathsf{Q}' \leftarrow \{h_1, \ldots, h_{i-1}\} \cup \{h'_i\} \cup \{h_{i+1} \ldots, h_q\}$
  // $\mathsf{Q}$ and $\mathsf{Q}'$ differ at exactly one point; at index $i$
$(i', \mathsf{aux}') \xleftarrow{\$} \mathcal{A}(X, \mathsf{Q}'; \rho)$
return $\perp$ if $i = \perp$
if $i = i'$
  return $(i, \mathsf{aux}, \mathsf{aux}')$
return $\perp$
```

Figure 2: Local Forking Algorithm

to the local forking lemma. We review the generalized forking lemma extension here, and the extension of the local forking lemma in Section 6.

Let $S \subseteq \{1, \ldots, q\}$. Let $\mathcal{A}$ be a randomized algorithm that on input $(X, \{h_1, \ldots, h_q\})$, outputs an index $i \in \{\perp\} \cup S$, as well as auxiliary output aux.

The probability that the adversary will complete successfully acc and the probability that $\mathsf{ForkAcc}$ will output success is then as in Equation 6.

$$\mathsf{ForkAcc} \geq \frac{\mathsf{acc}^2}{|S|} \tag{6}$$

Intuitively, Equation 6 gives (slightly) tighter bounds than the generalized forking lemma, as it bounds the acceptance probability to a subset $S \subseteq \{1, \ldots, q\}$. If it can be guaranteed that the adversary's output is strictly within S, then this variant allows for tighter bounds. Doing so is possible in the proof for FROST, as the adversary must query two random oracles in strict succession (as the output from the first random oracle is input into the second), which ensures that the environment can program the second random oracle at the time that the first is queried. Hence, the bounds can be scoped to the number of queries that the adversary has actually made to the first random oracle, as opposed to the range of allowed queries to the second random oracle.

6 Extension of Local Forking Lemma

We now review a "looser" local forking lemma again given by Bellare, Tessaro, and Zhu, this time in their analysis of the unforgeability for FROST.

The difference in this forking lemma is as follows. Here, forking occurs on two indices (i, j) in the first execution and (i', j') in the second execution. In this setting, i, i' are indices with respect to a random oracle. j is an auxiliary index that is guaranteed to be in the set J, and is used in the context of the wider proof. However, this forking lemma variant additionally enforces that the indices j, j are the same both before and after the fork, which results in a slightly looser bound than the original local forking lemma.

Why is this useful? This adaptation to the forking lemma is able to qualify over *more* information than simply some index i, as opposed to prior variants. In other words, additional auxiliary information can be reflected in the analysis of the adversary's likelihood of success.

This "looser" local forking lemma is given in the following algorithm.

```
Algorithm $\mathsf{ExtLFork}(X)$
Pick coins $\rho$ for $\mathcal{A}$ at random.
$\{h_1, \ldots, h_q\} \xleftarrow{\$} H^q$
$\mathsf{Q} \leftarrow \{h_1, \ldots, h_q\}$
$(i, j, \mathsf{aux}) \xleftarrow{\$} \mathcal{A}(X, \mathsf{Q}; \rho)$
  // j is guaranteed to be strictly in some set J
return $\perp$ if $i = \perp$
$h'_i \xleftarrow{\$} H$
$\mathsf{Q}' \leftarrow \{h_1, \ldots, h_{i-1}\} \cup \{h'_i\} \cup \{h_{i+1} \ldots, h_q\}$
$(i', j', \mathsf{aux}') \xleftarrow{\$} \mathcal{A}(X, \mathsf{Q}'; \rho)$
return $\perp$ if $i = \perp$
if $i = i'$ and $j = j'$
  return $(i, j, \mathsf{aux}, \mathsf{aux}')$
return $\perp$
```

The probability that $\mathsf{ForkAcc}$ will complete successfully is then reflected in Equation 7.

$$\mathsf{ForkAcc} \geq \frac{\mathsf{acc}^2}{q \cdot |J|} \tag{7}$$

7 Applications: Schnorr signatures

We now give an expanded version of the proof of security for Schnorr signatures. This proof is given by Bellare and Neven [2], but we show this proof here now with more detail.

7.1 Preliminaries

Definition 7.1. *A* ***signature scheme*** $\mathcal{SIG}$ *is a triple of PPT algorithms* $(\mathit{KeyGen}, \mathit{Sign}, \mathit{Verify})$ *invoked as*

$$(PK, sk) \xleftarrow{\$} \mathit{KeyGen}(1^\lambda), \quad \sigma \xleftarrow{\$} \mathit{Sign}(sk, m), \quad \mathit{Verify}(PK, m, \sigma) \in \{0, 1\}.$$

The scheme is **strongly unforgeable** *if the following function is negligible*

$$\mathsf{Adv}^{uf}_{\mathcal{A},\mathcal{SIG}}(\lambda) := \Pr\left[\begin{array}{l} \mathit{Verify}(PK, m, \sigma) = 1 \\ (m, \sigma) \notin \{(m_i, \sigma_i)\}_{i=1}^{q} \end{array} : \begin{array}{l} (PK, sk) \xleftarrow{\$} \mathit{KeyGen}(1^\lambda) \\ (m, \sigma) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{Sign}(\cdot)}(PK) \end{array}\right]$$

where $\mathcal{O}^{Sign}(m_i)$ *returns* $\sigma_i \xleftarrow{\$} \mathit{Sign}(sk, m_i)$ *for* $i = 1, \ldots, q$.

7.2 Schnorr Signatures

The Schnorr signature scheme is as follows, with respect to a hash function $H_c : (\mathbb{G}, \{0, 1\}^n, \mathbb{G}) \rightarrow \mathbb{Z}_p$.

- $\mathit{KeyGen}(1^\lambda)$: Sample a secret key $sk \xleftarrow{\$} \mathbb{Z}_p$; generate the public key as $PK \leftarrow g^{sk}$. Output (sk, PK).
- $\mathit{Sign}(sk, m)$: Sample a nonce $r \xleftarrow{\$} \mathbb{Z}_p$; generate its commitment as $R \leftarrow g^r$. Derive the challenge as $c \leftarrow H_c(PK, R, m)$ and the response as $z \leftarrow r + csk$. Output $\sigma = (R, z)$.
- $\mathit{Verify}(PK, \sigma = (R, z))$: Check that the equality $g^z \stackrel{?}{=} R \cdot PK^c$ holds. Output 0 if it does not, otherwise, output 1.

7.3 A Proof for Schnorr Signatures

We next show how to prove the security of Schnorr signatures using the general forking lemma. Note that the same result applies, even if the local forking lemma were instead employed.

Theorem 7.1. *For a group* $\mathbb{G}$ *of prime order* p *where the discrete logarithm problem is assumed to be hard, and hash function* $H_c(\mathbb{G}, \mathbb{G}, \{0, 1\}^*) \rightarrow \mathbb{Z}_p$*, the Schnorr signature scheme is unforgeable. In other words, for every adversary* $\mathcal{A}$ *that wins the unforgeability game against the Schnorr signature scheme, there exists an adversary* $\mathcal{D}$ *that uses the general forking algorithm* $\mathsf{GenFork}$ *as shown in Figure 1 to win the discrete logarithm game.*

Concretely, the advantage of $\mathcal{D}$ *is bounded by Equation 8.*

$$\mathsf{Adv}^{dl}_{\mathcal{SIG},\mathcal{D}}(\lambda) \geq \frac{(\mathsf{Adv}^{uf}_{\mathcal{SIG},\mathcal{A}}(\lambda))^2}{q} - \frac{2q_s}{p} - \frac{1}{p} \tag{8}$$

where $q = q_s + q_h + 1$*,* q_s *is the number of queries to the signing oracle that* $\mathcal{A}$ *is allowed to make, and* q_h *is the number of allowed random oracle queries.*

```
Algorithm $\mathcal{C}(X, \{h_1, \ldots h_q\})$
  // $q = q_s + q_h + 1$ is the total
  // number of possible queries
  // to $H_c$ during the simulation.
$PK \leftarrow X$
  // The challenge is the public key
$k \leftarrow 1$ // Track queries to random oracle
$k' \leftarrow 0$ // Track signing oracle queries
$\mathsf{bad} \leftarrow 0$
$E \leftarrow \emptyset, Q \leftarrow \emptyset$
$(m^*, \sigma^*) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^{Sign(\cdot)}}(PK)$
if $\mathsf{bad} = 1$
    return $\perp$ // fail if bad hash query
return $\perp$ if $(m^*, \sigma^*) \in E$
  // fail if signature is not a forgery
return $\perp$ if $\mathcal{SIG}.Verify(PK, m^*, \sigma^*) \neq 1$
  // fail if the forgery is not valid
$(R^*, z^*) \leftarrow \sigma^*$
$h_j \leftarrow Q(PK, R^*, m^*)$
  // challenge for $\mathcal{A}$'s forgery
$\mathsf{aux} \leftarrow (h_j, z^*)$
return $(j, \mathsf{aux})$
```

```
$\mathcal{O}^{Sign(m_i)}$
return $\perp$ if $k' \geq q_s$
  // $q_s$ is the total number
  // of allowed signing queries
$k' \leftarrow k' + 1$
$c_i \leftarrow h_k$
  // Get next hash value
$z_i \xleftarrow{\$} \mathbb{Z}_p$
$R_i \leftarrow g^{z_i} \cdot PK^{-c_i}$
if $Q[(PK, R_i, m_i)] \neq \perp$
  // $\mathcal{A}$ must guess $R_i$ with
  // $\frac{q}{2^\lambda}$ chance per query
    $\mathsf{bad} \leftarrow 1$
    return $\perp$
$Q[(PK, R_i, m_i)] = c_i$
  // Program random oracle
return $\sigma_i = (R_i, z_i)$
```

```
$H_c(PK, R_i, m_i)$
  // Performs lazy sampling
return $\perp$ if $k > q_H + 1$
  // $q_H$ is the total number
  // of allowed random oracle
  // queries, plus one forgery.
$Q[(PK, R_i, m_i)] = h_k$
$k \leftarrow k + 1$
return $h_k$
```

Figure 3: Algorithm $\mathcal{C}$ simulation of the unforgeability game for Schnorr signatures to an adversary $\mathcal{A}$. Without loss of generality, we assume $\mathcal{A}$ queries H_c on (PK, R^*, m^*).

We prove Theorem 7.1 by two lemmas.

Lemma 7.2. *There exists an algorithm $\mathcal{C}$ as shown in Figure 3. which can perfectly simulate the Schnorr signature scheme with respect to a challenge instance X.*

Proof. Let $\mathcal{A}$ be an adversary playing the EUF-CMA game against the Schnorr signature scheme, and let $\mathcal{C}$ be an algorithm that simulates this game to $\mathcal{A}$. We show the specifics of $\mathcal{C}$ in Figure 3.

Setup. $\mathcal{C}$ accepts as input the challenge instance $X \in \mathbb{G}$, and the set $q = q_h + q_s + 1$ values $\{h_1, \ldots, h_q\} \in \mathbb{Z}_p^q$. $\mathcal{C}$ sets its public key $PK = X$. $\mathcal{C}$ initializes tables E and Q to store its state and random oracle queries from $\mathcal{A}$. $\mathcal{C}$ then runs $\mathcal{A}$, providing as input PK.

$\mathcal{C}$ responds to $\mathcal{A}$'s signing and random oracle queries as follows.

Simulating Random Oracles. For each random oracle query by $\mathcal{A}$, $\mathcal{C}$ simulates the random oracles H_c by lazy sampling:

- $\underline{H_c(PK, R_i, m_i)}$: To begin, $\mathcal{C}$ checks to see if an entry in $Q[(PK, R_i, m_i)]$ exists. If so, $\mathcal{C}$ returns it. Else, it sets $Q[(PK, R_i, m_i)] = h_k$. It then returns h_k as its output.

Simulating Signing Oracles. $\mathcal{C}$ simulates the signing oracles to $\mathcal{A}$ as follows.

- $\underline{\mathcal{O}^{Sign(m_i)}}$: To begin, $\mathcal{C}$ assigns the challenge c_i as the next available hash value h_k. It then randomly samples the response $z_i \xleftarrow{\$} \mathbb{Z}_p$. It derives the commitment R_i as $R_i \leftarrow g^{z_i} \cdot PK^{-c_i}$, and then programs the random oracle as $Q(PK, R_i, m_i) = c_i$. It then returns (R_i, z_i) as its response.

$\mathcal{C}$'s simulation of $Sign(m_i)$ is perfect, because when the adversary verifies σ_i, it will first query $H_c(PK, R_i, m_i)$, and will receive c_i as output (because $\mathcal{C}$ has already programmed H_c on these inputs). The resulting signature will be valid, because of how R_i is derived with respect to z_i, PK and c_i

This completes the proof. □

Lemma 7.3. *Let $\mathcal{D}$ be an adversary playing against the discrete logarithm game, receiving as input a challenge $Y \in \mathbb{G}$ and winning the game if it produces a value x such that $Y = g^x$. The advantage of $\mathcal{D}$ is bounded by its ability to receive an accepting output from $\mathsf{GenFork}^{\mathcal{C}}$ and use this output to solve for the DL challenge, as characterized by Equation 8.*

Proof. We show the reduction $\mathcal{D}$ in Figure 4. To begin, $\mathcal{D}$ receives its discrete logarithm challenge X. The goal to $\mathcal{D}$ is to produce an output y such that $Y = g^x$.

$\mathcal{D}$ then queries $\mathsf{GenFork}^{\mathcal{C}}(X)$, where $\mathcal{C}$ is the simulating algorithm as defined in Figure 3. As output, $\mathcal{D}$ receives either $\perp$ indicating failure or $(1, \mathsf{aux}, \mathsf{aux}')$

```
Reduction $\mathcal{D}()$
1 : $X \xleftarrow{\$} \mathcal{O}^{\mathsf{DL}()}$
2 :   // Query for DL challenge
3 : $\mathsf{out} \xleftarrow{\$} \mathsf{GenFork}^{\mathcal{C}}(X)$
4 : if $\mathsf{out} = \perp$
5 :   return $\perp$
6 : $(1, \mathsf{aux}, \mathsf{aux}') \leftarrow \mathsf{out}$
7 : $(c^*, z^*) \leftarrow \mathsf{aux};\ (c^{**}, z^{**}) \leftarrow \mathsf{aux}'$
8 : $x \leftarrow \frac{z^* - z^{**}}{c^* - c^{**}}$
9 : return $x$
```

Figure 4: The reduction $\mathcal{D}$.

indicating success. Recall that $\mathsf{aux} = (c^*, z^*)$ and $\mathsf{aux}' = (c^{**}, z^{**})$. On failure, it returns $\perp$. Else, it returns the discrete logarithm x, by solving $x = \frac{z^* - z^{**}}{c^* - c^{**}}$.

The probability acc that $\mathcal{C}$ outputs an accepting answer is simply the advantage that $\mathcal{A}$ wins the unforgeability game, or:

$$\begin{aligned} \mathsf{acc} &= \Pr[\mathcal{C} \text{ outputs } 1] \\ &\geq \mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - \Pr[\mathsf{bad} = \mathsf{true}] \\ &\geq \mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - \frac{q_s q}{p} \end{aligned} \tag{9}$$

What is the probability that $\Pr[\mathsf{bad} = \mathsf{true}]$? Each time $\mathcal{A}$ queries the signing oracle, it has $\frac{q}{p}$ opportunities to guess R_i correctly (where $\mathbb{G}$ is of prime order p). So in total, $\Pr[\mathsf{bad} = \mathsf{true}] = \frac{q_s \cdot q}{p}$, because $\mathcal{A}$ is allowed q_s signing queries each time it is executed.

Recall that the codomain of $H_c(\mathbb{G}, \mathbb{G}, \{0,1\}^*) \to \mathbb{Z}_p$ is of order p. Hence, the probability that a hash collision occurs (i.e, $\Pr[\mathsf{HashCollision}]$) is $\frac{1}{p}$. Then, applying the General Forking Lemma 3.1, the advantage of $\mathcal{D}$ in solving its DL challenge is:

$$\begin{aligned} \mathsf{Adv}^{\mathrm{dl}}_{\mathcal{SIG},\mathcal{A}}(\lambda) &\geq \Pr[\mathsf{GenFork}^{\mathcal{C}}(X) \text{ outputs } 1] - \Pr[\mathsf{HashCollision}] \\ &\geq \mathsf{acc}(\frac{\mathsf{acc}}{q} - \frac{1}{p}) \quad \textit{// general forking lemma} \\ &\geq (\frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - q_s q)^2}{q} - \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - q_s q}{p} \end{aligned} \tag{10}$$

Now, observe that from Equation 9:

$$\begin{aligned} \mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - q_s q \leq 1, \text{ and so} \\ \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda)] - q_s q}{p} \geq -\frac{1}{p} \end{aligned} \tag{11}$$

Next, with some algebra, observe that:

$$\begin{aligned} (\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda))^2 &= (\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda))^2 - 2\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda)\frac{q_s q}{p} + (\frac{q}{p})^2 \\ &\geq (\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda))^2 - 2\frac{q_s q}{p} \end{aligned} \tag{12}$$

Hence, after some (more) algebra:

$$\begin{aligned} \mathsf{Adv}^{\mathrm{dl}}_{\mathcal{SIG},\mathcal{A}}(\lambda) &\geq (\frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - q_s q)^2}{q} - \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - q_s q}{p} \quad \textit{// From Equation 10} \\ &\geq \frac{(\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda) - q_s q)^2}{q} - \frac{1}{p} \quad \textit{// From Equation 11} \\ &\geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda)^2}{q} - \frac{\frac{2q_s q}{p}}{q} - \frac{1}{p} \quad \textit{// From Equation 13} \\ &\geq \frac{\mathsf{Adv}^{\mathrm{uf}}_{\mathcal{SIG},\mathcal{A}}(\lambda)^2}{q} - \frac{2q_s}{p} - \frac{1}{p} \end{aligned} \tag{13}$$

Which gives us Equation!8. This completes the proof. □

8 Conclusion

In this note, we review the first variant of the forking lemma presented by Pointcheval and Stern. We then review its generalization beyond signature schemes, and several subsequent variants. Each variant is tailored to the specific proof of security which it is employed, or allows for tighter bounds or generalization beyond signature schemes. We finally give an expanded variant of the proof for Schnorr signatures using the general forking lemma, and discuss how the local forking lemma can likewise be employed with slightly tigheter bounds.

9 Acknowledgements

Many thanks to Gregory Neven for his discussion on the proof for Schnorr signatures and notes on deriving the concrete bounds after applying the general forking lemma.

References

[1] M. Bellare, W. Dai, and L. Li. The local forking lemma and its application to deterministic encryption. In S. D. Galbraith and S. Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 607–636. Springer, 2019.

[2] M. Bellare and G. Neven. Multi-signatures in the plain public-key model and a general forking lemma. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security, CCS 2006, Alexandria, VA, USA, October 30 - November 3, 2006*, pages 390–399. ACM, 2006.

[3] D. Pointcheval and J. Stern. Security proofs for signature schemes. In U. M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 387–398. Springer, 1996.

[4] D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *J. Cryptol.*, 13(3):361–396, 2000.