# Concurrent Signatures

Samir Benzammour
*Algorithms and Complexity*
*RWTH Aachen University* Aachen, Germany
samir.benzammour@rwth-aachen.de

***Abstract*—By trying to solve the problem of fair signature exchange, Chen et al. proposed a concurrent approach that works for most real applications. It enables parties to produce signatures in an ambiguous way, regarding the parties identity, and are not bound by their signatures. They are bound to said signatures upon releasing an additional piece of information, the so-called *keystone*, after which the true identity of the signer will be known.**

## I. Introduction

Creating a signature scheme that provides true fairness in exchanging signatures is a well-studied problem and a few approaches already exist trying to solve it.

Among them is, for example, the optimistic fair exchange presented in [ASW98] with the usage of a trusted third party that can supervise the signature exchange and rule out any fraud on either side. The problem with this would be the need of a third entity, even if it is only needed when one party wants to cheat the other.

The concurrent signature scheme [CKP04] tries to solve that by using random bits to obfuscate the signing parties, who will be ambiguous to every other party. With this, neither party is at a disadvantage, e.g. if the other party does not want to sign the item anymore. These signatures are bound to their respective signer by releasing the key that generates said random bits, we call this key the *keystone*. This concurrent scheme does not provide a solution for a *truly fair* signature exchange, however, it provides an adequate solution for most applications.

In this paper, we will show how such a concurrent signature scheme can be constructed. The focus of this paper, however, lies in providing an intuitive understanding of the unforgeability proof provided in [CKP04]. We will make use of a concrete scheme defined in [CKP04] which builds upon other signature schemes. Note, that this signature scheme is not limited to the concrete one we define here, each suitable scheme can be used to produce a concurrent protocol.

In the following chapter, we will define certain preliminaries that we are going to need throughout this paper. Section III defines the concrete construction of this scheme and section IV describes how our construction should be used. Section V defines when our signature scheme can be labeled as *secure* and, lastly, section VI explains the unforgeability proof of [CKP04].

## II. Preliminaries

For our following construction and the proof, later on, we will need certain constructs and tools. These will be defined in the following.

***Definition 1** (Random Oracle):* A *random oracle* is a black-box algorithm which serves each *unique* request with a truly random value. Moreover, it serves each duplicate request with the same answer it sent before.

***Definition 2** (Chosen Message Attack):* In a signature scheme, a *chosen message attack* enables the adversary to get the signature of several messages, meaning it can query the signing oracle.

### II.1. Mathematical Concepts

In the following this chapter is going to introduce all mathematical concepts needed to follow the line of thought in the coming course of action.

***Definition 3** (Generator):* Let $\mathbb{G}$ be a cyclic group of order $p$. Then a group generator is $g \in \mathbb{G}$ such that it fulfills

$$\mathbb{G} = \{g^i \bmod p \mid i \in \mathbb{N}\}$$

This means, that by taking the generator to multiple powers, while taking the modulus of $p$, we can generate all elements of our given group.

***Definition 4** (Discrete Logarithm):* In a group $\mathbb{G}$, we define the *discrete logarithm* as an integer $x$ such that $b^x = a$ for an arbitrary integer $a$ and $b^x = \overbrace{b \cdot \ldots \cdot b}^{x \text{ times}}$.

***Definition 5** (Negligible function):* We say a function $f$ is negligible if

$$f(n) < \frac{1}{n^c}$$

for a sufficiently large $n \in \mathbb{N}$. This means, that our function $f$ converges towards 0.

### *II.2. Hardness of the Discrete Logarithm*

We define the hardness of the discrete logarithm analogously to [KL14], starting with the

***Definition 6** (Discrete logarithm experiment):* For this experiment $DLog_{\mathcal{A},\mathcal{G}}(\ell)$ we consider $\mathcal{G}$ to be a cyclic group generator that takes a security parameter $\ell$ as its input. Therefore, we start of by running $\mathcal{G}(1^\ell)$ to acquire $(\mathbb{G}, p, g)$ where $\mathbb{G}$ is a cyclic group of order $p$, and $g \in \mathbb{G}$ is a group generator of $\mathbb{G}$. We pick an $a \in \mathbb{G}$ at random, and pass all information to an algorithm $\mathcal{A}$. If $\mathcal{A}$ is able to calculate an $x$ such that $g^x = a$ we return 1 and 0 otherwise.

Continuing, now we define the actual hardness property we want to acquire

***Definition 7:*** We say that the *discrete logarithm problem is hard* relative to $\mathcal{G}$ if for all algorithms $\mathcal{A}$ there exists a negligible function negl such that

$$\Pr[DLog_{\mathcal{A},\mathcal{G}}(\ell) = 1] \leq \mathrm{negl}(\ell)$$

That means, that it is hard for an algorithm to reliably calculate the discrete logarithm of any number $a' \in \mathbb{G}$.

### *II.3. Forking Lemma*

In [PS96] and [PS00] Pointcheval and Stern introduced a tool for signature schemes, the *forking lemma*. It states that if an algorithm $E$ can produce a signature $\langle r_1, h, r_2 \rangle$ with $r_1$ has to be randomly chosen from a huge set, $h$ being the hash of the message and $r_1$, and $r_2$ depending only on $r_1, h$ and the message, on public input, then there exists another algorithm $\mathcal{A}$ that controls said algorithm $E$. $\mathcal{A}$ can then replay $E$, with replacing the signer by the simulation, to generate a second valid signature $\langle r_1, h', r_2' \rangle$ with $h \neq h'$.

### *II.4. Notation*

We define a certain set of notation to make communicating concepts easier.

- we define $\cdot \| \cdot$ as string concatenation
- we define $k$ to be be the keystone

## III. Construction

A signature scheme is called a *concurrent signature* scheme if it holds four algorithms, namely **SETUP**, **ASIGN**, **AVERIFY** and **VERIFY**. This concrete construction is based on the non-separable signature scheme in [AOS02], which, in turn, is a modification of the Schnorr signature scheme presented in [Sch91] with which we can leverage properties for our concurrent scheme. We define our algorithms as follows

### *III.1. **SETUP***

This algorithm initiates all relevant information for further course of action. It takes a security parameter $\ell$ as input and, among other things, returns two large primes $p$ and $q$ where $q \mid p-1$ has to hold. Furthermore, it returns a group generator $g \in (\mathbb{Z}/p\mathbb{Z})^*$ as well as $\mathcal{M}, \mathcal{S}, \mathcal{K}, \mathcal{F}$ with $\mathcal{M} \equiv \mathcal{K} = \{0,1\}^*$ and $\mathcal{S} \equiv \mathcal{F} = \mathbb{Z}_q$. It generates private keys $x_i$ which are randomly chosen from $\mathbb{Z}_q$, the public keys $\{X_i = g^{x_i} \bmod p\}$ and the set of participants $\mathcal{U}$. The public keys and the set of participants are available in the public domain. Continuing, two cryptographic hash functions $H_1, H_2 : \{0,1\}^* \to \mathbb{Z}_q$, alongside the function **KGEN** $:= H_1$ Lastly it returns any additional parameters $\rho$ that might be needed, and, note that, each participant keeps their respective private key $x_i$ hidden.

### *III.2. **ASIGN***

This algorithm takes $\langle X_i, X_j, x_i, f, M \rangle$ as input where $X_i, X_j$ are public keys with $i \neq j$. Furthermore $x_i \in \mathbb{Z}_q$ is the public key of $i$, $f \in \mathcal{F}$ being the keystone-fix, which is generated through **KGEN**, and $M \in \mathcal{M}$ being the message.

It calculates

$$h = H_2(g^t X_j^f \bmod p \parallel M)$$
$$h_1 = h - f \bmod q$$
$$s = t - h_1 x_i \bmod q$$

whereas $t \in \mathbb{Z}_q$ is chosen at random. With this it returns a, so called, *ambiguous signature* $\sigma = \langle s, h_1, f \rangle$.

### *III.3. AVERIFY*

This algorithm takes $S := \langle \sigma, X_i, X_j, M \rangle$ as input, with $\sigma = \langle s, h_1, f \rangle$. It returns $accept$ or $verify$ depending on whether or not

$$h_1 + f = H_2(g^t X_j^f \bmod p \parallel M) \bmod q \quad (1)$$

holds. This equation is checked because we have $h_1 = h - f$ and $h = H_2(g^t X_j^f \bmod p \parallel M)$. Therefore by rearranging it we get Equation 1.

### *III.4. VERIFY*

Lastly, this algorithm takes $\langle k, S \rangle$ as the input and checks whether or not **KGEN**$(k) = f$, and if this is the case, checks whether or not **ASIGN**$(S) = accept$. If both conditions hold, **VERIFY** returns $accept$ and $reject$ otherwise. To provide an overview for how each component works in **VERIFY**, Figure 1 shows exactly this synergy.

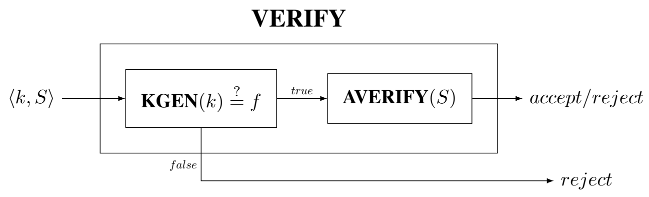

Fig. 1. **VERIFY** Components

## IV. Protocol

This section is responsible for providing an understanding of the concrete signature protocol and how the algorithms defined in section III should be used.

We will illustrate the protocol through an exemplary interaction between a party A(lice) and B(ob). The following description will be visually supported through Figure 2.

Firstly, we start by having both parties run the **SETUP** function to initialize all necessary variables for this scheme. Our, so called, *initial signer* continues with generating a keystone $k \in \mathcal{K}$ alongside the corresponding keystone-fix $f \in \mathcal{F}$. Keep in mind, that the keystone-fix is generated through **KGEN**$(k)$. We set, without loss of generality, Alice as our initial signer. Continuing, the initial signer starts off by signing a message $M_A \in \mathcal{M}$ through **ASIGN**, while providing the private keys of herself and Bob, her own private key $x_A$ and the keystone-fix $f$. Afterwards, Alice sends the resulting ambiguous signature $\sigma_A = \langle s_A, h_A, f \rangle$ to Bob. He verifies said signature by checking if **AVERIFY** returns $accept$, for which he additionally provides their respective public keys, as well as, Alice's message $M_A$. Note, that we are not trying to encrypt or obfuscate the messages in any way. Bob and Alice, repeat the signing and verification steps analogously for Bob with the *same keystone-fix*, i.e. Bob generates and sends his ambiguous signature with $f$ and Alice verifies it. Lastly, if Alice is satisfied with Bobs signature, she publishes the keystone $k$ with which both *ambiguous signatures* will be binding for both parties.

The power of who is generating and holding the keystone-fix is the one thing that keeps the concurrent signature scheme to be *truly fair*. Because with this, the initial signer (i.e. the one that generates the keystone(-fix)) can withhold the keystone. However, this would not be beneficial for the withholding party in real applications. Moreover, even if the withholding party keeps the keystone, without releasing it, the other party is not bound to the signature, meaning it is at no disadvantage.

## V. Security Model

To label our concurrent signature scheme as *secure* a certain set of conditions has to hold. In this scheme, we define these properties through fairness, ambiguity, and unforgeability.

To define the mentioned properties we, first of all, have to define our game, in which we have an adversary $E$ and a challenger $C$. In this game $C$ runs **SETUP** and publishes all public variables to the public domain.

The adversary can query the challenger for certain information.

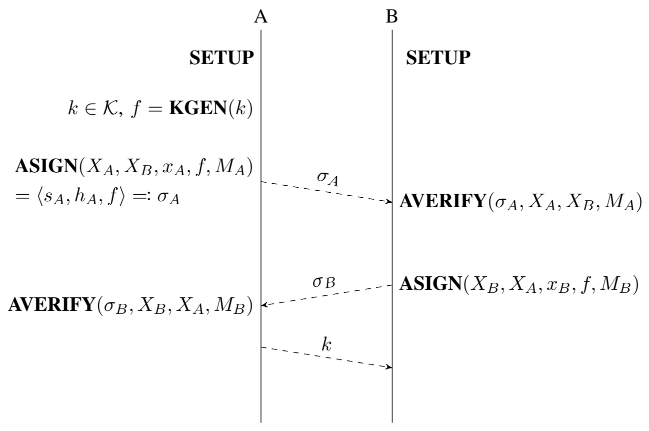
A
B
SETUP
SETUP
$k \in \mathcal{K}$, $f = $ **KGEN**$(k)$
**ASIGN**$(X_A, X_B, x_A, f, M_A)$
$= \langle s_A, h_A, f \rangle =: \sigma_A$
$\sigma_A$
**AVERIFY**$(\sigma_A, X_A, X_B, M_A)$
$\sigma_B$
**ASIGN**$(X_B, X_A, x_B, f, M_B)$
**AVERIFY**$(\sigma_B, X_B, X_A, M_B)$
$k$

Fig. 2. Concurrent Signature Protocol - An Overview

- the first being a **KGen** query, meaning $E$ can request $C$ to generate a keystone-fix
- it can also request the challenger to release the keystone to a given keystone-fix through a **KReveal** query.
- **ASign** queries achieve the goal to receive an *ambiguous signature* from the challenger on an adequate input.
- Lastly, the adversary can provide a public key for a **Private Key Extraction** query in order to receive the respective private key from $C$.

Note, that $C$ does not answer to **AVerify / Verify** queries because the adversary can run the needed algorithms itself.

## *V.1. Fairness*

In order to define fairness in our scheme, consider the previously defined game between the adversary and the challenger. The adversary can initially query everything it wants and $C$ answers accordingly. Eventually, however, the adversary returns two public keys $X_c, X_d$, a keystone $k \in \mathcal{K}$, $S = \langle \sigma, X_c, X_d, M \rangle$ with $\sigma = \langle s, h_1, f \rangle$, and **AVERIFY**$(S) = accept$. The adversary $E$ wins if either

1) if $f$ from previous query, no **KReveal** query was made on $f$ and if $\langle k, S \rangle$ is accepted by **VERIFY**, or
2) if $E$ produces a second $S' = \langle \sigma', X_d, X_c, M' \rangle$ with $\sigma' = \langle s', h'_1, f \rangle$ such that **AVERIFY**$(S') = accept$. Continuing, $\langle k, S \rangle$ is accepted by **VERIFY**, but $\langle k, S' \rangle$ is rejected.

In the first case, the adversary produces the keystone matching to an arbitrary $f$ which is valid. The second case describes the case in which the adversary is able to create a signature, with the same keystone-fix, that is not bound upon releasing the keystone.

## *V.2. Ambiguity*

Our defined game is split into three phases now, the first query phase, the challenge phase and, lastly, the second query phase. After the challenger sent all public variables to the adversary, the adversary starts the first query phase by requesting anything it wants but eventually, it has to send a, so-called, *challenge tuple* to the challenger. This tuple is of the form $\langle X_i, X_j, M \rangle$ and represents the info the adversary wants to be challenged to. Then, the challenger flips a coin $b \in \{0, 1\}$ and if $b = 0$ it sends an ambiguous signature $\sigma$ to the adversary on behalf of $X_i$. Otherwise, the challenger sends an ambiguous signature on behalf of $X_j$. Now, the adversary can start its second query phase, and eventually sends the challenge bit, which is a guess whose signature the challenger sent to our adversary. The adversary wins if the challenge bit matches the flipped coin of the challenger. Meaning, if the adversary was able to guess whose signature was sent, it managed to circumvent the ambiguity of the scheme.

## *V.3. Unforgeability*

We, again, consider the previously defined adversary–challenger game. The adversary can query everything it wants, but finally it has to return $S = \langle \sigma, X_c, X_d, M \rangle$ where $\sigma = \langle s, h_1, f \rangle$, $X_c, X_d$ being public keys and $M \in \mathcal{M}$ being the message. Continuing, the adversary wins the game if **AVERIFY**$(S) = accept$ and if either

1) No **ASign** query was made with input $\langle X_c, X_d, f, M \rangle$ or $\langle X_d, X_c, h_1, M \rangle$. And no **Private Key Extraction** query was made on either $X_c$ or $X_d$.
2) No **ASign** query was made with input $\langle X_c, X_i, f, M \rangle$ for all $X_i \neq X_c, X_i \in \mathcal{U}$ and

no **Private Key Extraction** query was made for $X_c$.

Semantically, the first case means that the adversary has no information about any of the public keys and tries to forge a signature for either $X_c$ and/or $X_d$. The second case, however, implies that the adversary has one private key (or at enough information to use it) and tries to forge a signature on behalf of $X_c$. This would imply that one of the parties wants to cheat the other.

### *V.4. Security of our scheme*

Now, that we defined our (concrete) concurrent signature scheme as well as the definition to when it can be called secure, we want to actually prove that it is. As mentioned before, this paper focuses on the unforgeability portion of the security proof, if one is interested in how the other properties are proven, reference [CKP04]. Knowing all of this, we define

***Lemma 1:*** The concrete concurrent signature scheme defined is existentially unforgeable under a chosen message attack in the random oracle model, assuming the hardness of the discrete logarithm problem.

## VI. Proof of Unforgeability

To start, we are going to describe the idea of the proof, which will dictate our further course of action

1.) *Assumption(s):* The discrete logarithm problem is hard.

2.) *Proof by contraposition:* We assume our scheme is not unforgeable.

3.) We construct an algorithm that leverages the forgeability of our scheme to contradict our assumption.

If our signature scheme is *not* unforgeable, there exists an algorithm $E$ that can forge a signature with non-negligible probability. Moreover, we define our algorithm $\mathcal{A}$ that solves the discrete logarithm as follows: $\mathcal{A}$ takes $\langle g, X, p, q\rangle$ as its input and uses $E$ as a component to solve the discrete logarithm problem, which would result in a contradiction to our assumption.

Before we start constructing our $\mathcal{A}$ we need to show that we can rewrite our ambiguous signature format $\sigma = \langle s, h_1, f\rangle$ as $\langle r_1, h, r_2\rangle$ in order to use the forking lemma later on. We define $r_1$ to be $g^t X_j^f \bmod p$, which fulfills the property that it takes its values randomly from a huge set $\mathbb{Z}_q$, $h = h_1 + f$ is the hash of the message $M$ and $r_1$, and lastly $r_2 = s$ which depends solely on $r_1, h$ and $M$.

Furthermore, we model our hash functions $H_1, H_2$ as random oracles, and in our following construction the algorithm $\mathcal{A}$ simulates these random oracles and the challenger $C$ that our forging-adversary $E$ plays against.

Now, our algorithm $\mathcal{A}$ starts by generating our participants $\mathcal{U}$. It guesses that our forging-adversary $E$ chooses $X_\alpha$ for the position $X_c$ in its output, and, therefore, sets $X_\alpha = X$. This means, that we want to acquire the discrete logarithm for the challenge public key $X_\alpha$. It also chooses a private key $x_i$, which is randomly chosen from $\mathbb{Z}_q$, for all participants $i$ that are not $\alpha$. Otherwise, the game would be pointless due to generating the discrete logarithm, we want to acquire, on our own. After all of that, $\mathcal{A}$ passes on $\langle g, p, q\rangle$, as well as the public keys, to our forging-adversary. As an overview of our setup, for now we have

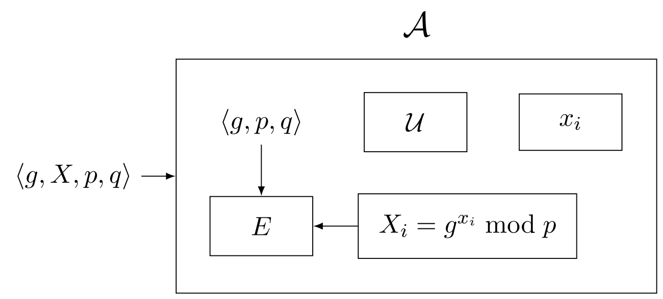

Fig. 3. Unforgeability Proof - Overview

Now, our forging-adversary $E$ can start querying requests, which include all of the queries from section V with a few additions and changes. We add $H_1$- and $H_2$ queries, with which $E$ can query the respective oracle to acquire a hash. Moreover, we modify our **ASign** queries such that if $\mathcal{A}$ receives a correct input $\langle X_i, X_j, f, M\rangle$ with $i \neq \alpha$ we return a signature as defined in section V. However, if $i = \alpha$ $\mathcal{A}$ picks random values for $h_1$ and $s$, and calculates the $H_2$ hash from these values. This happens with the premise that the random values and resulting string have never appeared in any $H_2$ query before, otherwise $\mathcal{A}$ chooses other values for $h_1$ and $s$ and starts again.

After a number of requests our forging-adversary returns an ambiguous signature $\sigma = \langle s, h_1, f \rangle$, public keys $X_c$ and $X_d$, and the message $M$. We know, from our definition in V-C, that the adversary wins if **AVERIFY** accepts $\sigma$ alongside with the public keys and the message, and if one of the following cases holds

1) No **ASign** query was made with input $\langle X_c, X_d, f, M \rangle$ or $\langle X_d, X_c, h_1, M \rangle$. And no **Private Key Extraction** query was made on either $X_c$ or $X_d$.
2) No **ASign** query was made with input $\langle X_c, X_i, f, M \rangle$ for all $X_i \neq X_c, X_i \in \mathcal{U}$ and no **Private Key Extraction** query was made for $X_c$.

In [AOS02] Abe et al. showed that our first case is negligible, assuming the hardness of the discrete logarithm problem. We know that our forging-adversary forges signatures with non-negligible probability, therefore, the second case must have occurred.

Furthermore, we assume that $X_c = X_\alpha = X$ because otherwise, $\mathcal{A}$ aborts as a result of failing to predict the correct the challenge public key.

Because **AVERIFY** returns *accept* and we know that it checks $h_1 + f \overset{?}{=} H_2(g^s X_c^{h_1} X_d^f \bmod p \parallel M)$, we now have the equation

$$h_1 + f \overset{?}{=} H_2(g^s X_c^{h_1} X_d^f \bmod p \parallel M)$$

which leaves two new cases to investigate.

The *first* case is, that $h = h_1 + f$ has never occurred in any signature query before. If that is the case, we rewrite our ambiguous signature to $\langle r_1, h, r_2 \rangle$ and, by using the forking lemma, force $E$ to produce a second signature $\langle r_1, h', r_2' \rangle$ with $h \neq h'$.

Now, we have two ambiguous signatures (by rewriting the second one) with $\sigma = \langle s, h_1, f \rangle$ and $\sigma' = \langle s', h_1', f' \rangle$, and $h \neq h'$. This results in $h = h_1 + f \neq h_1' + f'$ which implies that either $h_1 \neq h_1'$ or $f \neq f'$. The case in which $h_1 = h_1'$ is negligible because then the relevant $h_1$ queries would have to been calculated before the $H_2$ queries that produced $h$ and $h'$ which is not possible. In addition, it would imply that $f = f'$ but if that would be the case the output of our $H_1$ (that produces $f$ and $f'$) oracle would have to match an $H_2$ query. Henceforth, we know that $h_1 \neq h_1'$ and $f = f'$. As a consequence of $h$ and $h'$ resulting from different oracle queries we have

$$g^s X^{h_1} X_d^f = g^{s'} X^{h_1'} X_d^{f'} \tag{2}$$

$$s + xh_1 + f = s' + xh_1' + f' \quad \mid f = f' \tag{3}$$

$$s + xh_1 = s' + xh_1' \qquad \mid h_1 \neq h_1' \tag{4}$$

$$x = \frac{s - s'}{h_1' - h_1} \tag{5}$$

So, in the first case $\mathcal{A}$ solves the discrete logarithm problem of $X$ with non-negligible probability. This contradicts the hardness of the discrete logarithm problem and therefore, for the first case, this scheme is unforgeable.

Continuing with the second case, here we assume that $h = h'$ with $h' = h_1' + f$ appeared in a previous signature query. We denote said signature request with $\langle X_{c'}, X_{d'}, f', M' \rangle$ and its resulting signature with $\sigma' = \langle s', h_1', f' \rangle$. Because we have $h = h'$ we know that

$$g^s X^{h_1} X_d^f \bmod p = g^{s'} X_{c'}^{h_1'} X_{d'}^{f'} \bmod p$$

and $M = M'$ have to hold. Otherwise, the input of the hash functions would not be the same, and therefore $h \neq h'$.

Here, we have to differentiate further. If $X_{c'} = X$, $X_{d'} = X$ but their exponents differ, or $X_{c'}, X_{d'} \neq X$ we can easily derive the discrete logarithm $x$ of $X$ with the same argument as in the first case. It is important for the case where $X_{c'} = X$ or $X_{d'} = X$, that the exponents differ because otherwise we would have a division by 0 in step 4 of resolving to $x$.

However, let's take a closer look at the case where $X_{c'} = X$ or $X_{d'} = X$, and their exponents are *the same*. If we had $X_{c'} = X$ and $h_1 = h_1'$, because the exponents (i.e. $h_1, h_1'$) are the same, then that would imply that $f = f'$ due to $h = h'$. This, however, is a contradiction to the condition that no **ASign** queries can be made of the form $\langle X_c, X_i, f, M \rangle$ because we set $X_c = X$. Now, what if he had $X_{d'} = X$ and $h_1 = f'$. Then, when $h_1' = h' - f'$ was generated in a previous **ASign** query, $f = h_1'$ already had to hold by definition. Moreover, we also know that the output of **ASign** queries is determined by oracles $H_1$ and $H_2$. Now, the problem is that $f$ also is generated by $H_1$. That means that an output of $H_1$ has to match an $h_1'$, that is generated by $H_2$ and $H_1$. The probability of this occurring is negligible and therefore not relevant.

With this, we showed that the discrete logarithm problem could also be solved in the second case. Globally speaking, this means that lemma 1 holds.

## VII. Conclusion

To conclude, in this paper we illustrated the concept of concurrent signatures defined by Chen et al., and provided a rundown of the unforgeability proof provided in [CKP04].

In short, the concurrent signature scheme enables parties to sign a contract ambiguously, in respect to their own identity, until another piece of information is released. This piece of information is called the keystone and is generated by one of the participating parties. To add, the generation of the keystone by one of the participating parties is also the reason why the concurrent signature scheme does not solve the problem with the *truly fair* signature exchange. However, the presented solution is an adequate enough solution for most applications.

Lastly, keep in mind that the unforgeability property is not a sufficient condition for the security of this scheme. To see how the other two properties are proven, reference [CKP04], or if you are interested on how this scheme is making use of its underlying signature structure, see [AOS02] and [Sch91].

## References

[AOS02] Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. 1-out-of-n signatures from a variety of keys. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 415–432. Springer, 2002.

[ASW98] Nadarajah Asokan, Victor Shoup, and Michael Waidner. Optimistic fair exchange of digital signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 591–606. Springer, 1998.

[CKP04] Liqun Chen, Caroline Kudla, and Kenneth G Paterson. Concurrent signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 287–305. Springer, 2004.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[PS96] David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 387–398, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[PS00] David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of cryptology*, 13(3):361–396, 2000.

[Sch91] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.