Minimizing Trust in Hardware Wallets with Two Factor Signatures

Antonio Marcedone[1], Rafael Pass[*1], and abhi shelat[†2]

[1]Cornell Tech, {marcedone,rafael}@cs.cornell.edu
[2]Northeastern University, abhi@neu.edu

January 2, 2019

Abstract

We introduce the notion of *two-factor signatures (2FS)*, a generalization of a two-out-of-two threshold signature scheme in which one of the parties is a *hardware token* which can store a high-entropy secret, and the other party is a *human* who knows a low-entropy password. The security (unforgeability) property of 2FS requires that an external adversary corrupting either party (the token or the computer the human is using) cannot forge a signature.

This primitive is useful in contexts like hardware cryptocurrency wallets in which a signature conveys the authorization of a transaction. By the above security property, a hardware wallet implementing a two-factor signature scheme is secure against attacks mounted by a malicious hardware vendor; in contrast, all currently used wallet systems break under such an attack (and as such are not secure under our definition).

We construct efficient provably-secure 2FS schemes which produce either Schnorr signature (assuming the DLOG assumption), or EC-DSA signatures (assuming security of EC-DSA and the CDH assumption) in the Random Oracle Model, and evaluate the performance of implementations of them. Our EC-DSA based 2FS scheme can directly replace currently used hardware wallets for Bitcoin and other major cryptocurrencies to enable security against malicious hardware vendors.

1 Introduction

Cryptocurrency hardware wallets are increasingly popular among Bitcoin and Ethereum users as they offer seemingly stronger security guarantees over their software counterparts. A hardware wallet is typically a small electronic device (such as a USB device with an input button) that holds the secret key(s) to one or more cryptocurrency "accounts". It provides a simple interface that can be used by client software on a computer or smartphone to request a signature on a particular transaction; the wallet returns a signature to the client if the user has authorized it by pressing the physical button[1]. Typically, the user also has to enter a pin or password, either on the device itself or through the client. Some hardware wallets like the Trezor include a screen that can be used by the user to confirm the details of the transaction before authorizing it.

Ideally, a hardware wallet runs a firmware that is smaller and simpler than the software running on a common laptop (and thus may be less vulnerable to bugs and exploits), is built using tamper proof hardware that makes it difficult to directly read its memory, and is designed to prevent the private keys it holds from ever leaving the device. Thus, stealing funds from an address controlled by a hardware wallet is considered to be harder than stealing from a software wallet installed on the user's laptop.

[*] Supported in part by NSF Award CNS-1561209, NSF Award CNS-1217821, NSF Award CNS-1704788, AFOSR Award FA9550-15-1-0262, AFOSR Award FA9550-18-1-0267, a Microsoft Faculty Fellowship, and a Google Faculty Research Award.

[†] Supported in part by NSF grants 1664445 and 1646671.

[1] The physical button prevents malware from abusing the wallet without cooperation from the user.

Can we trust the hardware manufacturer? However, most hardware wallets suffer from a serious issue: since the wallet generates and holds the secret keys for the user's account, a compromised wallet might be used to steal the entirety of the coins it controls. Consider, for instance, a malicious wallet manufacturer who introduces a backdoored pseudorandom generator (to be used, for example, to generate the signing keys) into a hardware wallet. Because of the tamperproof properties of the hardware, such a backdoor might be extremely hard to detect and go unnoticed even to a scrupulous user, especially if it only affects a small portion of the company's devices (perhaps those shipped to customers who hold large coin balances). Yet, without the need of ever communicating with the devices again, the manufacturer might suddenly steal all the money controlled by those addresses before anyone has time to react! This is also true in the case where the user picks a password to supplement the entropy generated by the backdoored PRG, since passwords have limited entropy which can be bruteforced and, as we detail later, the wallet can bias the randomness in the signatures to leak information about such password.

Even if the company producing the wallet is reputable and trusted, supply chain attacks by single employees or powerful adversaries are still hard to rule out for customers. For example, the NSA reportedly intercepts shipments of laptops purchased online in transit to install malware/backdoors [18]. Indeed, trust in a wallet manufacturer, its supply chain, and the delivery chain are a serious concern.

One possible solution is to store the funds in a multi-signature account controlled by a combination of hardware (and possibly software) wallets from different manufacturers. However, the above is inconvenient and limiting. It may also be possible for a single supplier to corrupt multiple manufacturers of hardware wallets.

A Formal Treatment of Hardware Wallets In this paper, we initiate a formal study of the security of hardware wallets. As discussed above, completely relying on the token to perform key generation and signing operations requires a strong trust assumption on the hardware manufacturer. To avoid this, we focus on a scenario in which the user has both a single *hardware token* and a (low-entropy) *password*, and formally define appropriate an appropriate cryptographic primitive, which we name *two factor signature scheme (2FS)*.

Roughly speaking, a 2FS scheme can be thought of a special type of two-out-of-two threshold signature scheme [4] but where one of the parties (the user) only has a (potentially low-entropy) password, whereas the other party (the hardware token) can generate and store high-entropy secrets. Even defining unforgeability properties of such 2FS schemes turns out to be a non-trivial task; we provide the first such definitions. Our notions of unforgeability consider both malicious clients, malicious tokens, and attackers that may have selective access to honestly implemented tokens.

As already mentioned, as far as we know, in all currently known/used schemes, unforgeability does not hold when the hardware token can be maliciously designed, and thus no currently known schemes satisfies even a subset of our unforgeability definitions. Our main contribution is next the design of 2FS that satisfy them. In fact, we present a general transformation from any two-out-of-two threshold signature scheme which satisfies some additional technical property—which we refer to as *statistical Non-Signalling*—into a 2FS in the random oracle model, which produces public keys and signatures of the same form as the underlying threshold signature scheme.

We note that it may be possible to generically modify any TS to become Non-Signalling by having the parties perform coin-tossing to generate the randomness, and then prove in zero knowledge that they executed the signing protocol consistently with the pre-determined (and uniform) randomness. Using such a method, however, would result in a (polynomial-time but) practically inefficient scheme. In contrast, in the full version of this work [13], we show how to adapt two existing threshold signature schemes to satisfy this new technical property with very little overhead. Using our transformation, this gives secure 2FS schemes which efficiently generate Schnorr and ECDSA signatures.

Theorem (Informal). Assuming the discrete logarithm assumption, there exists a secure 2FS scheme in the Random Oracle model which generates Schnorr Signatures.

Theorem (Informal). Assuming the DDH assumption holds and that EC-DSA is unforgeable, there exists a secure 2FS scheme in the Random Oracle model that generates EC-DSA signatures.

The first construction is based on the Schnorr TS signature scheme of Nicolosi et al [15], while the second one is a slight modification of an EC-DSA threshold scheme of Lee *et al.* [5]. As EC-DSA signature are currently used in Bitcoin, Ethereum and most other major crypto currencies, our 2FS for EC-DSA can be directly used for hardware wallets supporting those crypto currencies. To demonstrate its practicality, we evaluate such scheme and estimate its performance on hardware tokens that are much less powerful than the CPUs on which we can benchmark the protocol. We confirm that running the protocol on two server-class CPUs (Intel) requires roughly 3ms to sign a message. When one of the parties is run on a weak computer (e.g., a Raspberry Pi 3b) and the other is run on a server, the protocol requires roughly 50ms. Our estimates confirm that the bottleneck in our scheme will be the processing capacity of the hardware token. Using a very secure, but weak 8-bit 1Mhz ATECC family processor [14], we estimate that ECDSA keys can be produced in under a minute and signatures can be completed in 3s. The entire signing process requires human input to complete (button press), and thus is likely to take seconds overall anyway.

1.1 Technical overview

The Definition. At a high level, in a Two Factor Signature scheme the signatures are generated by two parties: a client C who receives a (typically low entropy) password as input from a user, and a token T, which can store and generate secrets of arbitrary length, can produce signatures for multiple public keys and as such keeps a state which can be modified to add the ability to sign for new public keys. It consists of a tuple of algorithms $(\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$, where $\mathbf{KeyGen_T}(1^\kappa, s_T)$ and $\mathbf{KeyGen_C}(pwd)$ are an interactive protocol used by the token and client respectively to produce a public key and to accordingly update the token state s_T by "adding a share of the corresponding secret key"; $\mathbf{PK_C}(pwd)$ and $\mathbf{PK_T}(s_T)$ are two algorithms used by the client and the token (on input the password and the current token state s_T respectively) interacting with each other to retrieve a public key pk which was previously generated using the first two algorithms; $\mathbf{Sign_C}(pwd, m)$ and $\mathbf{Sign_T}(s_T, m)$ are similarly used to produce signatures; $\mathbf{Ver}(pk, m, \sigma)$ is used to verify the signatures.

We proceed to outline the unforgeability properties we require from such Two Factor Signature scheme. We consider 4 different attack scenarios, and define "best-possible" unforgeability properties for each of them. The first two are simply analogs of the standard unforgeability (for "party 1" and "party 2") properties of two-out-of-two threshold signatures.

1. *For the Client*: The simplest and most natural attack scenario is when the user's laptop is compromised (i.e. by malware), even before the key generation phase. We require that, except with negligible probability, such an adversary cannot forge signatures on a message m with respect to a public key which the token outputs (and would typically show to the user on its local screen) unless it asked the token to sign m. This notion mirrors the classic one of unforgeability (for party 1) of threshold signature schemes.

2. *For the Token*: We next consider an attack scenario in which the adversary can fully control the token T. We let it interact arbitrarily with an honest client, and receive the signatures and public keys output by such client during these interactions. We require that the probability that such an adversary can produce a forgery on a message m that would verify with respect to one of the public keys output by the client (during a **KeyGen** execution) without asking the client to sign m, is bounded by the min-entropy of the user's password. Again, this notion mirrors the classic notion of unforgeability of threshold signatures (for party 2), except that since the user only has a low-entropy password, we cannot require the probability of forging to be negligible; instead, we bound it by $q/2^m$ where q is the number of random oracle queries performed by the adversary, and m is the min-entropy of the password distribution.

Note that the unforgeability for the token security bound is rather weak (when the password has low entropy), but is necessarily so because the only secret held by the client is the password, and thus an attacker that "fully controls the token" (i.e., controls its input/outputs while at the same time participating in other outside interaction) and gets to see public keys, can simply emulate the client algorithm with a guessed password and attempt to create a forgery. Yet, note that to carry out this type of attack (which leads to the "unavoidable" security loss) and profit from it is quite non-trivial in practice as it requires the token to be able to somehow communicate with an attacker in the outside world (which is challenging given that a hardware wallet is a physically separate entity without a direct network connection).

Consequently, we consider two alternative attack scenarios that leverage the fact that often the token cannot communicate with the adversary and capture more plausible (i.e weaker) attack models. Yet, in these weaker attack models, we can now require the forging probability bounds to be significantly stronger.

3. *For the Token Manufacturer*: We consider an adversary who cannot fully control the T party, but can specify ahead of time a program Π which the T party runs. For example, this models the case of a malicious token manufacturer who embeds a PRG with a backdoor. Program Π can behave arbitrarily, but its answers to the interactions with any client have to satisfy the correctness properties of the scheme with overwhelming probability (if the token aborted or caused the client to return signatures which do not verify w.r.t. the expected public keys, the user could easily identify such token as faulty or malicious). The adversary can then have an honest client interact arbitrarily with Π ($\mathcal{A}$ is given the resulting public keys and signatures), and should not be able to produce a forgery on a message m that would verify with respect to one of the public keys output by the client (during a **KeyGen** execution) unless it received a signature on m as a result of such an interaction. We require the forging probability to be negligible (as opposed to bounded by $q/2^m$).

4. *With Access to the Token*: An alternative scenario is one where the token is not corrupted, but the attacker can get access to it (for example, in the case of a lost/stolen token, or a token shared between multiple users). More precisely, the adversary can interact with an honest T and may also interact with an honest client C (which itself interacts with T) and has to produce forgeries on a message m (which C did not sign, but on which T can be queried) w.r.t. a public key which C output during an interaction with T. Whereas unforgeability for the token implies that the above-mentioned adversary's forging probability is bounded by $q/2^m$ where q is the number of random oracle queries, we here sharpen the bound to $q'/2^m$ where q' is the number of invocations of T. (As T could rate-limit its answers by e.g., 1 sec, q' will be significantly smaller than q in practice.)

As far as we know, no previously known scheme satisfies all of the the above properties; in fact, none satisfy even just (1) and (2), or (1) and (3).[2]

The Construction The high-level idea behind our construction is natural (although the approach is very different from Trezor and other currently used hardware wallets). We would like to employ a two-out-of-two threshold signature (TS) scheme where the token is one of the parties and the client is the other. The problem is that the client only has a low-entropy password and cannot keep any persistent state. In fact, even if it had a high-entropy password, it wouldn't be clear how to directly use the threshold schemes as in general (and in particular for EC-DSA), secret key shares for threshold schemes are generated in a correlated way.

To overcome this issue, the key generation algorithm begins by running the key generation procedure for the TS: the token and the client each get a secret key share (which we denote sk_T and sk_C respectively), as well as the public key pk. Next, since the client cannot remember pk, sk_C, it encrypts pk, sk_C using a key that is derived—by using a random oracle (RO)—from its password; additionally, the client generates (deterministically) a random "handle" as a function of its password, again by applying the RO to the password. It then sends both the handle and the (password-encrypted) ciphertext to the token for storage.

Later on, when a client wants to get a signature on a message m, it first asks the token to retrieve its password-encrypted ciphertext: the token will only provide it if the client provides the correct handle (which the honest client having the actual password can provide). Next, the client decrypts the ciphertext (again using the password), and can recover its public and secret key. Finally, using its secret key, and interacting with the token the client can engage in the threshold signing process to obtain the desired signature on m.

The Analysis: Exploiting Non-Signalling and Exponential-time Simulation While we can show that the above construction satisfies properties 1,2 and 4 assuming the underlying threshold scheme is secure, demonstrating property 3—that is, security against malicious token manufacturers, which in our opinion is the most cruicial property—turns out to be non-trivial.

[2] Although we are not aware of any formal analysis of Trezor, it would seem that it satisfies (1) and (4), but there are concrete attacks against the other properties.

The issue is the following: as already mentioned, if the token is fully controlled by the attacker (which participates in outside interactions), then we can never hope to show that unforgeability happens with negligible probability as the attacker can always perform a brute-force attack on the password. In particular, in our scheme, the attacker can simply brute-force password guesses against the ciphertext c to recover the client's threshold secret key share. However, a malicious manufacturer which generates a malicious token but cannot directly communicate with it, would have more trouble doing so. Even if the malicious token program can perform a brute-force attack, it cannot directly communicate the correct password (or the client key share) to the manufacturer! If the token could somehow signal these information to the manufacturer, then the manufacturer could again break the scheme. And in principle, with general threshold signatures, there is nothing that prevents such signalling. For example, if the token could cause the threshold signing algorithm to output signatures whose low-order bits leak different bits of c, after sufficiently many transactions that are posted on a blockchain, the adversary could recover c and brute force the password himself.

Towards addressing this issue, we define a notion of *Non-Signalling* for TS: roughly speaking, this notion says that even if one of the parties (the token) is malicious, as long as they produce accepting signatures (with overwhelming probability), they cannot bias the distribution of the signatures generated—i.e., such signatures will be indistinguishable from honestly generated ones. In fact, to enable our proof of security—which proceeds using a rather complex sequence of hybrid arguments relying on *exponential-time simulation*—we will require the TS scheme to satisfy a *statistical* notion of Non-Signalling which requires that the distribution of signatures generated interacting with the malicious party is statistically close to the honest distribution.

We next show that if the underlying TS indeed satisfies statistical Non-Signalling, then our 2FS also satisfies property 3. Towards doing this, we actually first show that our 2FS satisfies an analogous notion of Non-Signalling, and then show how to leverage this property to prove unforgeability for the token manufacturer. We mention that the notion of Non-Signalling for 2FS is interesting in its own right: it guarantees that a maliciously implemented token Π (whose answers are restricted to satisfy the correctness properties of the scheme with overwhelming probability) cannot leak (through the public keys and the signatures which it helps computing) to an attacker any information which an honestly implemented token would not leak. In particular, if the honest token algorithm generates independent public keys and uses stateless signing (as the ones we consider do), even a malicious token cannot leak correlations between which public keys it has been used to create, or what messages it has signed.

1.2 Additional Properties

Unlinkability Other than the Non-Signalling property, our scheme satisfies an additional desirable *privacy* property, *Unlinkability*, which roughly speaking guarantees that multiple public keys (and signatures with respect to different public keys) generated with the help of the same honest token (or with different tokens on input the same password) look independent from each other as if they were generated by both different tokens and passwords. This property holds even an adversary who can interact arbitrarily with such tokens, as long as it cannot guess the passwords. This allows a user to have different public keys using the same hardware token with the guarantee that such public keys will look unrelated to each other to any external observer. Further, even if a token is lost or such user has to go through customs and her hardware wallet is inspected, a public key cannot be linked to such token unless the corresponding password is known.

Backing up the token state If the token is lost, stolen or otherwise malfunctioning (due to hardware failures, or even "ransom attacks" where the manufacturer might set up the token to stop working unless a ransom is paid), the user might not be able to produce signatures any more, which would result in a loss of money. Most current hardware wallet solutions derive all the secret keys for the addresses they sign for from a single short seed (say 256 bits), and offer the possibility to backup the seed at initialization time, by encoding it as a list of a few words that are displayed directly on the token's secure display for the user to write down [16]. This ensures that the seed is never stored on the user's laptop and thus is out of reach from malware.

In our case, the token's state is much larger (as it includes randomness generated from the client and a ciphertext encrypted under the user's password), and therefore this method cannot be applied directly.

As an alternative solution, the token might encrypt its whole state at initialization time with a fresh key, transfer the ciphertext to the client's computer (so that it can be backed up), and give the key to the user directly (again, in the form of a few words shown on the token's screen for the user to write down). In this case, extra care should be taken to ensure that a malicious token cannot leak information through the ciphertext in the case where such ciphertext is later compromised (for example by choosing a key known to the manufacturer or using biased randomness in the encryption). For this reason, the client should also re-encrypt the ciphertext under a fresh key of its own choice (which the user should again write down and store securely). We stress that both in this case and in existing hardware wallet solutions, correctness of the backup is not guaranteed. As such, these backups should be verified using independent devices.

Handling multiple addresses efficiently Another important feature for hardware wallet users is to be able to control multiple independent looking key pairs (addresses). Briefly, using several addresses rather than a single one to store one's currency makes tracking multiple transactions made by the same user more difficult. For example in Bitcoin, when a transaction is performed, the "change" generated by the transaction is often moved to a fresh address (controlled by the sender) rather than being sent back to the address which is funding the transaction. While it is feasible to use more than one public key with the same hardware wallet with our proposed Two Factor Signature scheme, by just running the **KeyGen** algorithm every time a new address is needed, this is inconvenient because each such new "change" address generation requires a separate backup step. Indeed, most commercial hardware wallets offer the possibility to derive more than one signing key pair from the same seed.

One first approach to overcome this limitation would be to extend the scheme by generating more than one independent key pair for the underlying threshold signature scheme every time the key generation algorithm of the Two Factor Signature scheme is run. These different addresses could be distinguished by a counter (which the user would provide as an additional parameter to the **Sign** and **PK** algorithms to identify which of the key pairs to use). Moreover, they would all be encrypted and backed up together with a single encryption key (using the mechanism of the previous paragraph) so that the number of words the user has to write down doesn't increase. As an additional optimization, in our EC-DSA based instantiation, the most computationally expensive output of the key generation algorithm (namely the precomputed OT extensions) could be reused accross multiple key pairs for increased efficiency.

Another possibility is to pick a threshold signature scheme that allows to efficiently sign w.r.t. multiple independent looking public keys given a single pair of secret key shares (whose size does not depend on the number of public keys). The EC-DSA based threshold scheme we describe can be modified in this sense by applying a technique detailed in section 4.4 of [8] (this idea can be adapted to the Schnorr case as well). Briefly speaking, in the original scheme, at the end of the **KeyGen** algorithm each party holds a multiplicative share of the secret signing key. It is therefore easy to modify such algorithm so that the client and the token can agree on an additional random string c (which is stored by the token and included by the client in the ciphertext it sends to the token). Then, as an additional input to the **Sign** algorithm, the client can pick any identifier i to distinguish each of the many public keys required (for example, a simple counter): the public key associated with identifier i would be computed by both parties as $pk_i = pk^{H(c,pk,i)}$ (where pk is the original public key and H is a hash function), and signatures could be computed by having the client use $sk_{C,i} = sk_C \cdot H(c, pk, i)$ instead of sk_C as the secret key in the threshold signature algorithm (the token would keep using the same secret key). See [8] for more details. Note that these modifications to the signature scheme would require adapting the security definitions to fit this scenario, which we leave to future work. For example, the Non-Signalling definition would also have to require that the many public keys generated by different values of i look independent to an outside observer or to the token manufacturer. Similarly, the unforgeability definitions and proofs would need to be adapted to ensure that a corrupted client cannot use an execution of the signature algorithm where the token intends to produce a signature for the key associated with counter i to produce a signature for a different identifier i' (we conjecture that this is the case for the EC-DSA signature scheme we propose).

1.3 Related Work

Threshold Signatures Threshold signatures [4, 7, 17, 2, 1] are signature schemes distributing the ability to generate a signature among a set of parties, so that cooperation among at least a threshold of them is required to produce a signature. Nicolosi *et al.* [15] present a threshold signature scheme for the Schnorr signature scheme. Particularly relevant to the cryptocurrency application are the works of Goldfeder *et al.* [8, 6], Lindell [9, 10], and Lee *et al.* [5] which propose a threshold signature scheme to produce ECDSA signatures, which is already compatible with Bitcoin and Ethereum.

Passwords + Threshold signatures MacKenzie and Reiter [11, 12] and Camenish *et al.* [3] consider notions somewhat similar to the one of a password-based threshold signature scheme: as in our setting, signing requires knowledge of a password and access to an external party (in their case a server rather than a hardware token), but in contrast to our setting the signer may additionally hold some *high-entropy secret state* (and indeed, the schemes considered in those papers require such secret state). This rules out the usage of such schemes in our scenario, as we want the user to be able to operate his wallet from any client without relying on any external state beyond its password.

1.4 Organization of the paper

Section 2 introduces some notations, recalls the security definitions for encryption schemes and presents some auxiliary lemmas. In Section 3 we recall the definition of Threshold Signature scheme and introduce the Non-Signalling property. Section 4 defines Two Factor Signature schemes and Section 5 presents our main construction and proofs of security. Section 6 presents the two modified TS schemes (based on Schnorr and EC-DSA) which can be used to instantiate our construction, and Section 7 discusses the Unlinkability property.

2 Preliminaries and Notation

If X is a probability distribution, we denote with $x \leftarrow X$ the process of sampling x according to X. When, in a probabilistic experiment, we say that an adversary outputs a probability distribution, we mean that such a distribution is given as a poly-time randomized program such that running the program with no input (and uniform randomness) samples from such distribution. For two party (randomized) algorithms we denote with $\langle \alpha; \beta \rangle \leftarrow \langle A(a); B(b) \rangle$ the process of running the algorithm A on input a (and uniform randomness as needed) interacting with algorithm B on input b (and uniform randomness), where α is the local output of A and β is the local output of B. Whenever an algorithm has more than one output, but we are interested in only a subset of such outputs, we will use $\cdot$ as a placeholder for the other outputs (for example we could write $(\cdot, pk) \leftarrow \mathbf{KeyGen}(1^\kappa)$ to denote that pk is a public key output by the **KeyGen** algorithm of a signature scheme in a context where we are not interested in the corresponding secret key).

Token Oracles. In our definitions, we will often model a party/program implementing party T. We say that a Token Oracle is a stateful oracle which can answer **KeyGen**, **PK**, **Sign** queries. Initially, its state is set to $\perp$. To answer such queries, the oracle interacts with its caller by running the $\mathbf{KeyGen_T}$, $\mathbf{PK_T}$, $\mathbf{Sign_T}$ algorithms respectively using its own inner state (and a message m supplied by the caller for **Sign** queries). As a result of $\mathbf{KeyGen_T}$ queries, its state is also updated. Moreover, when explicitly specified, the oracle could also return to the caller the public keys pk which are part of its local output during $\mathbf{KeyGen_T}$ and $\mathbf{Sign_T}$ queries.

2.1 Symmetric encryption

Our construction requires indistinguishability under a chosen ciphertext attack: no adversary, given an encryption oracle that, each time it is queried, takes as input two messages and always encrypts the first or the second one, and a decryption oracle which cannot be queried on outputs of the first oracle, should be able to tell which of the two messages the encryption oracle is encrypting. We also require integrity of ciphertexts: no adversary, given an encryption oracle, should be able to create a new ciphertext which successfully decrypts.

Definition 1 (Symmetric Encryption)**.** *A Symmetric Encryption scheme is a triple of PPT algorithms* $SE = (\mathbf{SE.G}, \mathbf{SE.E}, \mathbf{SE.D})$ *such that:*

- $\mathbf{SE.G}(1^\kappa) \to ek$ *is the "Key Generation algorithm", which on input the security parameter* κ *outputs a secret key* ek*. In this work, we assume without loss of generality that* $\mathbf{SE.G}$ *simply samples*[3] $ek \leftarrow_R \{0,1\}^\kappa$*.*
- $\mathbf{SE.E}(ek, m) \to c$ *is the "Encryption algorithm", which on input a secret key* ek *and a message* m *outputs a ciphertext* c*.*
- $\mathbf{SE.D}(ek, c) \to m$ *is the "Decryption algorithm", which on input a secret key* ek *and a ciphertext* c *outputs a message* m*, or a special error symbol* $\perp$ *if the ciphertext is invalid.*

We require that any Symmetric Encryption scheme satisfies the following correctness property for all κ *and all messages* m*:*

$$\Pr[ek \leftarrow \mathbf{SE.G}(1^\kappa) : \mathbf{SE.D}(ek, \mathbf{SE.E}(ek, m)) = m] = 1.$$

Definition 2 (Indistinguishability of plaintexts)**.** *Let* $\mathit{SE} = (\mathbf{SE.G}, \mathbf{SE.E}, \mathbf{SE.D})$ *be a Symmetric Encryption scheme. Consider the following the following experiment between an adversary* $\mathcal{A}$ *and a challenger, parameterized by a bit* b*:*
$\mathbf{SE.ExpIND\text{-}CCA}^{\mathit{SE}}_{\mathcal{A},b}(1^\kappa)$*:*

1. *The challenger computes* $ek \leftarrow \mathbf{SE.G}(1^\kappa)$*.*
2. *It runs* $\mathcal{A}(1^\kappa)$*.* $\mathcal{A}$ *can ask* $\mathbf{SE.E}$ *and* $\mathbf{SE.D}$ *queries. For any* $\mathbf{SE.E}$ *query,* $\mathcal{A}$ *submits two messages* m_0, m_1 *of the same size and receives* $\mathbf{SE.E}(ek, m_b)$*. For any* $\mathbf{SE.D}$ *query on input* c*,* $\mathcal{A}$ *receives* $\mathbf{SE.D}(ek, c)$ *if* c *is not the output of a previous* $\mathbf{SE.E}$ *query, and* $\perp$ *otherwise.*
3. *The adversary outputs a bit* b'*, which defines the output of the experiment.*

A Symmetric Encryption scheme SE *is said to be* ***IND-CCA secure*** *if for all PPT* $\mathcal{A}$ *there exists a negligible function* μ *such that*

$$|\Pr[\mathbf{SE.ExpIND\text{-}CCA}^{\mathit{SE}}_{\mathcal{A},0}(1^\kappa) = 1] - \Pr[\mathbf{SE.ExpIND\text{-}CCA}^{\mathit{SE}}_{\mathcal{A},1}(1^\kappa) = 1]| < \mu(\kappa)$$

Definition 3 (Integrity of ciphertexts)**.** *Let* $\mathit{SE} = (\mathbf{SE.G}, \mathbf{SE.E}, \mathbf{SE.D})$ *be a Symmetric Encryption scheme. Consider the following the following experiment between an adversary* $\mathcal{A}$ *and a challenger, parameterized by a bit* b*:*
$\mathbf{SE.ExpINT\text{-}CTXT}^{\mathit{SE}}_{\mathcal{A},b}(1^\kappa)$*:*

1. *The challenger computes* $ek \leftarrow \mathbf{SE.G}(1^\kappa)$*.*
2. *It runs* $\mathcal{A}(1^\kappa)$*.* $\mathcal{A}$ *can ask* $\mathbf{SE.E}$ *and* $\mathbf{SE.D}$ *queries. For any* $\mathbf{SE.E}$ *query,* $\mathcal{A}$ *submits a message* m *and receives* $\mathbf{SE.E}(ek, m)$*. For any* $\mathbf{SE.D}$ *query on input* c*,* $\mathcal{A}$ *receives* $\mathbf{SE.D}(ek, c)$ *if* c *is not the output of a previous* $\mathbf{SE.E}$ *query, and* $\perp$ *otherwise.*
3. *When* $\mathcal{A}$ *halts, the output of the experiment is defined to be 1 if* $\mathcal{A}$ *received an output* $m \neq \perp$ *from at least one* $\mathbf{SE.D}$ *query, and 0 otherwise.*

A Symmetric Encryption scheme SE *is said to be* ***INT-CTXT secure*** *if for all PPT* $\mathcal{A}$ *there exists a negligible function* μ *such that*

$$\Pr[\mathbf{SE.ExpINT\text{-}CTXT}^{\mathit{SE}}_{\mathcal{A}}(1^\kappa) = 1] < \mu(\kappa)$$

For our case, a Symmetric Encryption scheme which is both IND-CCA and INT-CTXT secure can be constructed in the random oracle model.

[3] One can turn any encryption scheme into one which satisfies this convention by considering as a secret key the randomness used in the key generation algorithm. When using an encryption scheme based on a block cipher such as AES, this requirement is already fulfilled.

2.2 Min Entropy

Definition 4 (Min Entropy)**.** *Let X be a discrete random variable. The min entropy of X is*

$$H_\infty(X) = \max_{x \in X} - \log_2 \Pr[X = x]$$

The following are two useful lemmas which bound the probability of guessing a password with the min entropy of the distribution it is sampled from.

Lemma 5. *Consider the following experiment involving an adversary $\mathcal{S}$:*
$\mathbf{ExpGuess}_{\mathcal{S}}(1^\kappa)$:

1. *$\mathcal{S}(1^\kappa)$ outputs a distribution* `PWD` *over $\{0,1\}^\kappa$.*

2. *The challenger samples pwd ←* `PWD` *and continues running the adversary $\mathcal{S}^{\textbf{\textit{Guess}}(\cdot)}$, which now has access to an oracle which takes as input a guess pwd' and returns 0 or 1 depending on whether $pwd = pwd'$.*

3. *When $\mathcal{S}$ halts, the output of the experiment is set to be* 1 *if one of the guesses from $\mathcal{S}$ was correct, and* 0 *otherwise.*

For all integer functions $q(\kappa), m(\kappa)$, all (even computationally unbounded) adversaries $\mathcal{S}$ which output distributions whose min entropy is lower bounded by $m(\kappa)$ and which make at most $q(\kappa)$ queries, it holds that

$$\Pr[\mathbf{ExpGuess}_{\mathcal{S}}(1^\kappa) = 1] \leq \frac{q(\kappa)}{2^{m(\kappa)}}$$

Proof. For any fixed number of guesses $q(\kappa)$, the adversary with the best success probability is the one which guesses the $q(\kappa)$ passwords which have the greatest probability of being sampled by the distribution `PWD`. By the definition of min entropy, each of these guesses will be correct with probability at most $2^{-m(\kappa)}$, from which the claim follows by a union bound. □

Lemma 6. *Consider the following game involving an adversary $\mathcal{S}$:*
$\mathbf{ExpGuessTwo}_{\mathcal{S}}(1^\kappa)$:

1. *$\mathcal{S}(1^\kappa)$ outputs a distribution* `PWDS` *over $\{0,1\}^\kappa \times \{0,1\}^\kappa$.*

2. *The challenger samples (pwd_0, pwd_1) ←* `PWDS` *and continues running the adversary $\mathcal{S}^{\textbf{\textit{Guess}}(\cdot)}$, which now has access to an oracle which takes as input a guess pwd' and returns 1 if $pwd = pwd_0$ or $pwd = pwd_1$ and 0 otherwise.*

3. *When $\mathcal{S}$ halts, the output of the experiment is set to be* 1 *if one of the guesses from $\mathcal{S}$ was correct, and* 0 *otherwise.*

For each distribution `PWDS` *over $\{0,1\}^\kappa \times \{0,1\}^\kappa$, we can consider the two "truncated" distributions over $\{0,1\}^\kappa$ obtained by sampling an element from* `PWDS` *and truncating the first or the last κ bits.*

For all integer functions $q(\kappa), m(\kappa)$, all (even computationally unbounded) adversaries $\mathcal{S}$ which output distributions where each of the truncations has entropy lower bounded by $m(\kappa)$ and make at most $q(\kappa)$ guesses, it holds that

$$\Pr[\mathbf{ExpGuessTwo}_{\mathcal{S}}(1^\kappa) = 1] \leq \frac{2q(\kappa)}{2^{m(\kappa)}}$$

Proof. The proof is similar to the previous one. For any fixed number of guesses $q(\kappa)$, the adversary with the best success probability is the one which guesses the $q(\kappa)$ passwords which have the greatest probability of being sampled as the first or second element by the distribution `PWDS`. By the definition of min entropy, each password can be sampled with probability at most $2 \cdot 2^{-m(\kappa)}$, as it can be sampled with the same probability of at most $2^{-m(\kappa)}$ both as the first and as the second component of the couple. The claim follows by a union bound. □

3 Threshold Signature scheme

This section recalls the definition of a Threshold Signature scheme. In addition to the standard Unforgeability properties, we also introduce a new *Non-Signalling* property which is required by our construction. The formalization presented here is for a 2-party setting (C and T) and the key shares are computed by the parties using a distributed key generation algorithm (as opposed to being provided by a trusted dealer).

Definition 7. *A (2-out-of-2) Threshold Signature scheme consists of a tuple of distributed PPT algorithms defined as follows:*

- $\langle \mathbf{TS}.\mathbf{GC}(1^\kappa); \mathbf{TS}.\mathbf{GT}(1^\kappa)\rangle \rightarrow \langle sk_C, pk; sk_T, pk\rangle$ *are two randomized algorithms which take as input the security parameter and, after interacting with each other, produce as output a public key pk (output by both parties) and a secret key share for each of them. We use* $\mathbf{TS}.\mathbf{Gen}(1^\kappa) \rightarrow (sk_C, sk_T, pk)$ *as a compact expression for the above computation.*

- $\langle \mathbf{TS}.\mathbf{SC}(sk_C, m); \mathbf{TS}.\mathbf{ST}(sk_T, m)\rangle \rightarrow \langle \sigma; \perp\rangle$ *are two randomized algorithms interacting to produce as output a signature*[4] σ*. We use* $\mathbf{TS}.\mathbf{Sign}(sk_C, m, sk_T) \rightarrow \sigma$ *as as a compact expression for the above computation.*

- $\mathbf{TS}.\mathbf{Ver}(pk, m, \sigma) \rightarrow 0 \vee 1$ *is a deterministic algorithm. It takes as input a public key, a message and a signature and outputs 1 (accept) or 0 (reject).*

These algorithms have to satisfy the following correctness property: for all messages m

$$\Pr\left[\begin{array}{c}(sk_C, sk_T, pk) \leftarrow \mathbf{TS}.\mathbf{Gen}(1^\kappa):\\ \mathbf{TS}.\mathbf{Ver}(pk, m, \mathbf{TS}.\mathbf{Sign}(sk_C, m, sk_T)) = 1\end{array}\right] = 1$$

Definition 8. *Let TS be a Threshold Signature scheme. Define the following experiment between an adversary $\mathcal{A}$ and a challenger:*
$\mathbf{TS}.\mathbf{ForgeC}_{\mathcal{A}}^{TS}$*:*

1. *The challenger executes* $\mathbf{TS}.\mathbf{GT}(1^\kappa)$ *interacting with $\mathcal{A}$ (who plays the role of C) and obtains a public key pk and a secret key share sk_T. It gives pk to $\mathcal{A}$.*
2. *The adversary can adaptively ask signing queries for an arbitrarily chosen message m. For each such query, the challenger runs* $\mathbf{TS}.\mathbf{ST}(sk_T, m)$ *interacting with $\mathcal{A}$ in the role of C.*
3. *The adversary outputs (m, σ). The output of the experiment is 1 if* $\mathbf{TS}.\mathbf{Ver}(pk, m, \sigma) = 1$ *but the adversary never asked a signing query for m.*

An analogous experiment **TS.ForgeT** *can be defined, where the adversary plays the role of C and the challenger plays the role of T. In this case, for each signing query the adversary also gets the signature locally output by the* $\mathbf{Sign_C}$ *algorithm which the challenger runs.*

A TS is said to be existentially ***Unforgeable for*** C *(resp.* ***Unforgeable for*** T*) under a chosen message attack if for all PPT $\mathcal{A}$ the probability that* **TS.ForgeC** *(resp.* **TS.ForgeT***) outputs 1 is negligible. A TS which satisfies both properties is simply said* ***Unforgeable****.*

The *Non-Signalling* definition consists of two properties. First, we require that a malicious token cannot bias the distribution of the public keys output by **TS.Gen** when interacting with an honest client (as long as such token does not make the **TS.Gen** execution abort). More in detail, we require that for any polynomial sized circuit Π (which does not make the execution of **TS.Gen** abort with more than negligible probability), the distribution of public keys output by an execution of the **TS.GC** interacting with Π in the role of T is statistically indistinguishable from the distribution obtained by running **TS.Gen** with both parties implemented honestly. This is formalized as an experiment where an adversary $\mathcal{A}$ (not necessarily running

[4] This definition states that party T does not output the signature. However, in our construction we do not rely on σ being "hidden" from T, so threshold schemes where both parties learn the signature can also be used in our construction.

in polynomial time) outputs a PPT program Π and then has to distinguish whether it is given a public key generated by an honest client interacting with Π or by an honest client interacting with an honest token.

Analogously, the second property requires that a malicious token cannot bias the distribution of signatures output by the **TS.Sign** algorithm. An adversary $\mathcal{A}$ outputs a public key pk, a message m, a secret key for the client sk_C and a polynomial sized circuit Π which can interact with a client running $\mathbf{TS.SC}(sk_C, m)$, such that (with all but negligible probability) the output for the client interacting with Π is a valid signature on m w.r.t. pk. We require that $\mathcal{A}$ cannot distinguish between the output of such an interaction and a valid signature on m w.r.t. pk sampled uniformly at random.

Definition 9. *Let* $\mathit{TS} = (\mathbf{TS.GC}, \mathbf{TS.GT}, \mathbf{TS.SC}, \mathbf{TS.ST}, \mathbf{TS.Ver})$ *be a Threshold Signature scheme. Consider the following two experiments between an adversary* $\mathcal{A}$ *and a challenger, each parameterized by a bit* b*:*

$\mathbf{TS.NS1}_{\mathcal{A}}^{\mathit{2FS},b}(1^\kappa)$:

1. $\mathcal{A}(1^\kappa)$ *outputs a polynomial size (in* κ*) circuit* Π*, such that* $\Pr[\langle\cdot, pk; \cdot\rangle \leftarrow \langle \mathbf{TS.GC}(1^\kappa); \Pi\rangle : pk \neq \perp] > 1 - \mu(\kappa)$ *(i.e. running the circuit interacting with an honest* **TS.GC** *implementation results in such honest implementation outputting* $\perp$ *with at most negligible probability).*

2. *If* $b = 0$*, the challenger computes* $\langle\cdot, pk; \cdot\rangle \leftarrow \langle \mathbf{TS.GC}(1^\kappa); \Pi\rangle$*; otherwise it computes* $\langle\cdot, pk; \cdot\rangle \leftarrow \langle \mathbf{TS.GC}(1^\kappa); \mathbf{TS.GT}(1^\kappa)\rangle$*. Then it returns pk to* $\mathcal{A}$*.*

3. $\mathcal{A}$ *outputs a bit* b'*, which defines the output of the experiment.*

$\mathbf{TS.NS2}_{\mathcal{A}}^{\mathit{2FS},b}(1^\kappa)$:

1. $\mathcal{A}(1^\kappa)$ *outputs a polynomial size (in* κ*) circuit* Π*, a secret key share* sk_C*, a message* m *and a public key* pk*, such that* $\Pr[\langle\sigma; \cdot\rangle \leftarrow \langle \mathbf{TS.SC}(sk_C, m); \Pi\rangle : \mathbf{TS.Ver}(pk, m, \sigma) = 1] > 1 - \mu(\kappa)$ *(i.e. running the circuit interacting with an honest* **TS.SC** *implementation on input* sk_C, m *results in such honest implementation outputting a valid signature for* m *under* pk *with overwhelming probability).*

2. *If* $b = 0$*, the challenger computes* $\langle\sigma; \cdot\rangle \leftarrow \langle \mathbf{TS.SC}(1^\kappa); \Pi\rangle$*; otherwise it samples a valid signature at random, i.e. it samples* $\sigma \leftarrow_R \{\sigma : \mathbf{Ver}(pk, m, \sigma) = 1\}$*. Then it returns* σ *to* $\mathcal{A}$*.*

3. $\mathcal{A}$ *outputs a bit* b'*, which defines the output of the experiment.*

TS is said to be ***Non-Signalling*** *if for all PPT adversaries* $\mathcal{A}$ *there exist a negligible function* μ *such that*

$$|\Pr[\mathbf{TS.NS1}_{\mathcal{A}}^{\mathit{2FS},0}(1^\kappa) = 1] - \Pr[\mathbf{TS.NS1}_{\mathcal{A}}^{\mathit{2FS},1}(1^\kappa) = 1]| < \mu(\kappa)$$
$$|\Pr[\mathbf{TS.NS2}_{\mathcal{A}}^{\mathit{2FS},0}(1^\kappa) = 1] - \Pr[\mathbf{TS.NS2}_{\mathcal{A}}^{\mathit{2FS},1}(1^\kappa) = 1]| < \mu(\kappa)$$

If the above equations hold even for adversaries $\mathcal{A}$ *which are not bounded to be PPT (but that output circuits* Π *which still have to be polynomially sized), the TS is said to be* ***Statistically Non-Signalling****.*

4 Two Factor Signature Schemes

A Two Factor Signature scheme is similar to a 2-out-of-2 threshold signature scheme, where signatures are generated by two parties: a client C whose only long term state is a (typically low entropy and independently generated) password, and a token T, who can store and generate secrets of arbitrary length. We envision the token party T to be implemented on a hardware token (which a user would carry around) with a dedicated screen and button which would ask the user for confirmation before producing signatures.

The semantics of the scheme are designed to capture the fact that a token party T has a single state s_T which can be used as input to produce signatures according to different public keys (for which an initialization phase was previously performed). This is useful, as typically a hardware wallet would offer support for multiple cryptocurrency accounts, and therefore such semantics allow us to design a scheme which natively supports multiple such accounts and reason about the security of the whole system.

More specifically, one can think of each public key that the scheme can produce signatures for as being associated with both a password and a (not necessarily private) mnemonic key identifier (or account identifier in the hardware wallets application) chosen by the user (i.e.“`savings`” or “`vacation_fund`”). In order to generate a new public key the client executes the **KeyGen** algorithm with a token T. The client's inputs are the key identifier and its password *pwd*, while the token updates its state s_T as a result of running this algorithm. Later, the client can produce signatures for that public key on a message m by running the **Sign** algorithm (interacting with the same token) on input m and the same password and key identifier. Additionally, the **PK** algorithm can be used to reconstruct a previously generated public key (both the password and the key identifier are required in this case as well). In our formal description, for the sake of simplicity and w.l.o.g., we consider such key identifier to be part of the password itself.

Definition 10. *A Two Factor Signature scheme (2FS) consists of a tuple of PPT algorithms:*

- $\langle \mathbf{KeyGen_C}(pwd); \mathbf{KeyGen_T}(s_T)\rangle \rightarrow \langle pk; pk, s'_T\rangle$ *are two randomized algorithms interacting with each other to produce as output a public key pk (output by both parties).* s_T *represents the state of party* T *before running the algorithm (which would be* $\perp$ *on the first invocation), and* s'_T *represents its new updated state. We use* $\mathbf{KeyGen}(pwd, s_T) \rightarrow (pk, s'_T)$ *as a compact expression for the above computation.*

- $\langle \mathbf{PK_C}(pwd); \mathbf{PK_T}(s_T)\rangle \rightarrow \langle pk; pk\rangle$ *are two algorithms interacting with each other to produce as output a public key. We use* $\mathbf{PK}(pwd, sid, s_T) \rightarrow pk$ *as a compact expression for the above computation.*

- $\langle \mathbf{Sign_C}(pwd, m); \mathbf{Sign_T}(s_T, m)\rangle \rightarrow \langle \sigma; \perp\rangle$ *are two randomized algorithms interacting with each other to produce as output a signature* σ*, output by the first party only. We use* $\mathbf{Sign}(pwd, m, s_T) \rightarrow \sigma$ *as as a compact expression for the above computation.*

- $\mathbf{Ver}(pk, m, \sigma) \rightarrow 0 \vee 1$ *is a deterministic algorithm. It takes as input a public key, a message and a signature and outputs 1 (accept) or 0 (reject).*

These algorithms have to satisfy the following correctness properties. Let s_T *be any valid token state (i.e. any state obtained by starting with* $\perp$ *as the initial state and then updating it through several executions of* **KeyGen** *on input arbitrary passwords), pwd be any password which was used in at least one such execution of* **KeyGen***, pk be the output of the* $\mathbf{KeyGen_C}$ *algorithm in the most recent of the executions of* **KeyGen** *on input pwd. We require that both*

$$\Pr[\mathbf{PK}(pwd, s_T) = pk] = 1,\ \Pr[\mathbf{Ver}(pk, m, \mathbf{Sign}(pwd, m, s_T)) = 1] = 1$$

As discussed in the introduction, in order to give precise security guarantees depending on the capabilities of the adversary, we formalize several unforgeability notions for a 2FS.

Unforgeability for the Client The first property, *Unforgeability for the Client*, is framed as an experiment where the adversary plays the role of a client interacting with an oracle simulating an honestly implemented token. The adversary is allowed to interact with the token by adaptively asking multiple **KeyGen**, **PK**, **Sign** queries on arbitrary inputs. The definition requires that (except with negligible probability) after interacting n times with the token oracle on **Sign** queries for a specific message m, the adversary cannot produce signatures on m for more than n of the public keys that the token had output during **KeyGen** queries. Moreover, it requires that the adversary cannot make the token output (as the result of a **PK** query) a public key which the token did not help generating (and thus did not output during a previous **KeyGen** query).

In order to check the conditions described above, the challenger keeps a list g of all the public keys returned to $\mathcal{A}$ by a **KeyGen** query, a list p of the public keys returned by a **PK** query, and for any message m it records how many **Sign** queries for m were asked by $\mathcal{A}$.

Definition 11. *Let 2FS* $= (\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$ *be a Two Factor Signature scheme. Consider the following experiment between an adversary* $\mathcal{A}$ *and a challenger:*
$\mathbf{ExpForgeC}_{\mathcal{A}}^{\mathit{2FS}}(1^\kappa)$:

1. *The challenger runs the adversary* $\mathcal{A}$*, giving it access to a token oracle* $\mathbf{T}$ *(*$\mathcal{A}$ *is given the pk values output by such oracle during* **KeyGen** *and* **PK** *queries).* $\mathcal{A}$ *can interact with the oracle arbitrarily. In addition, the challenger records the pk values locally output by the token oracle for* **KeyGen** *queries on an (initially empty) list g, and for* **PK** *queries on an (initially empty) list p.*
2. $\mathcal{A}$ *halts and outputs a message m and a list of forgeries* $(pk_1, \sigma_1), \ldots, (pk_n, \sigma_n)$*. We define the output of the experiment as 1 if either there exists a pk that belongs to p but not to g, or if for all* $i \in \{1, \ldots, n\}$*,* $\mathbf{Ver}(pk_i, m, \sigma_i) = 1$*, all the* pk_i *are distinct and are in g, and* $\mathcal{A}$ *made at most* $n - 1$ **Sign** *queries to the oracle* $\mathbf{T}$ *on input m.*

2FS is said to be ***Unforgeable for the Client*** *if for all PPT adversaries* $\mathcal{A}$ *there exist a negligible function* μ *such that for all* κ

$$\Pr[\mathbf{ExpForgeC}_{\mathcal{A}}^{\mathit{2FS}}(1^\kappa) = 1] \leq \mu(\kappa).$$

Unforgeability for the Token The second property, *Unforgeability for the Token*, is analogous to the previous one. It is formalized as an experiment where $\mathcal{A}$ plays the role of a token who can interact in arbitrary $\mathbf{KeyGen_C}$,$\mathbf{PK_C}$,$\mathbf{Sign_C}$ queries with an honest client simulated by the challenger. The definition requires that (except as specified below) after interacting n times with the client oracle on **Sign** queries for a specific message m, the adversary cannot produce signatures on m for more than n of the public keys that the client had output during **KeyGen** queries. Moreover, it requires that the adversary cannot make the client output (as a result of a $\mathbf{PK_C}$ query) a public key which the client did not help generating (during a $\mathbf{KeyGen_C}$ query).

In this case, since we want the honest client not to keep any long term state between different executions besides the password (so that having physical access to the hardware token and remembering the password is enough to produce signatures), the definition must allow the adversary to succeed with the same probability with which it can guess such password. This is formalized by presenting the definition in the random oracle model. Initially, the adversary outputs a password distribution PWD from which the challenger samples the password which the honest client will use. The scheme is unforgeable for the token if no adversary can forge with probability better than the one of guessing such password (given a number of guesses equal to the number of random oracle queries asked by $\mathcal{A}$). We quantify this probability in terms of the min entropy of the password distribution PWD.

Definition 12. *Let 2FS* $= (\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$ *be a Two Factor Signature scheme. Consider the following experiment between an adversary* $\mathcal{A}$ *and a challenger:*
$\mathbf{ExpForgeT}_{\mathcal{A}}^{\mathit{2FS}}(1^\kappa)$:

1. *The challenger runs the (stateful) adversary* $\mathcal{A}(1^\kappa)$ *on input the security parameter.* $\mathcal{A}$ *outputs a distribution PWD over* $\{0,1\}^\kappa$*.*
2. *The challenger samples* $pwd \leftarrow_R$ *PWD.*
3. *The adversary can adaptively query the following oracles:*
   - $\mathbf{KeyGen_C}$*: It takes no input from* $\mathcal{A}$*, and executes* $\mathbf{KeyGen_C}(pwd)$ *interacting with* $\mathcal{A}$ *(who plays the role of T) and gives* $\mathcal{A}$ *its local output pk. In addition, the challenger appends pk to an (initially empty) list g.*
   - $\mathbf{PK_C}$*: This oracle takes no input from* $\mathcal{A}$*, and executes* $\mathbf{PK_C}(pwd)$ *interacting with* $\mathcal{A}$ *and also gives it its local output pk. In addition, the challenger appends pk to an (initially empty) list p.*
   - $\mathbf{Sign_C}$: *It takes as input m from* $\mathcal{A}$*, and runs* $\mathbf{Sign_C}(pwd)$ *interacting with* $\mathcal{A}$*, and also gives it its local output* σ*.*

- $\mathbf{RO}_\kappa(\cdot)$. *This oracle implements a random oracle which might be required by the scheme.*

4. *$\mathcal{A}$ halts, and possibly outputs a message m and a tuple of forgeries $(pk_1, \sigma_1), \ldots, (pk_n, \sigma_n)$. We define the output of the experiment as 1 if either there exists a pk that belongs to p but not to g, or if for all $i \in \{1, \ldots, n\}$, $\mathbf{Ver}(pk_i, m, \sigma_i) = 1$, all the pk_i are distinct and are in g, and the $\mathbf{Sign_C}$ oracle was queried at most $n-1$ times on m.*

2FS is said to be ***Unforgeable for the Token*** *if for all PPT adversaries $\mathcal{A}$ such that $\mathcal{A}(1^\kappa)$ makes at most $q(\kappa)$ queries to the random oracle and outputs distributions PWD with min entropy lower bounded by $m(\kappa)$, there exist a negligible function μ such that for all κ*

$$\Pr[\mathbf{ExpForgeT}_{\mathcal{A}}^{2FS}(1^\kappa) = 1] \leq \frac{q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa).$$

Unforgeability With Access to the Token In the next definition, *Unforgeability With Access to the Token*, we consider an adversary which can arbitrarily interact with a token oracle, and in addition has access to a client oracle. This client oracle models a client who uses a password sampled from a distribution chosen by the adversary and answers the adversary's queries by interacting with the token oracle. Unlike in previous definitions, here the adversary is allowed to interact with the token on input the message on which it will forge, but its success probability in forging is bound by the probability of guessing the password (given one "guess" per interaction with the token oracle). As before, this limitation is expressed in terms of the min entropy of the distribution from which the password is sampled.

Definition 13. *Let 2FS = $(\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$ be a Two Factor Signature scheme. Consider the following experiment between an adversary $\mathcal{A}$ and a challenger:*
$\mathbf{ExpForgePwdGuess}_{\mathcal{A}}^{2FS}(1^\kappa)$:

1. *The challenger runs the adversary $\mathcal{A}$, giving it access to a token oracle $\mathbf{T}$ ($\mathcal{A}$ is also given the pk values output by such oracle during* **KeyGen** *and* **PK** *queries). $\mathcal{A}$ can interact with the oracle arbitrarily.*

2. *During the execution, $\mathcal{A}$ outputs a distribution PWD. The challenger samples pwd $\leftarrow$ PWD. From this point on (in addition to $\mathbf{T}$), the adversary is allowed to make queries to a new client oracle $\mathbf{C}$ which supports $\mathbf{KeyGen_C}$, $\mathbf{PK_C}$, $\mathbf{Sign_C}$ queries. For each query, this oracle will execute the respective client algorithm on input pwd, interact with the token oracle $\mathbf{T}$ (by asking an appropriate query) and return $\mathbf{C}$'s local output to $\mathcal{A}$ (for $\mathbf{Sign_C}$ queries, $\mathcal{A}$ can also specify the message m that the oracles will use in their interaction).*

   In addition, for each $\mathbf{KeyGen_C}$ query where the $\mathbf{C}$'s output given to the adversary is pk, the challenger creates a record (pk, s) where s is an initially empty set of messages and adds it to an (initially empty) list g. For each $\mathbf{Sign_C}$ query on input pwd and m, the challenger adds m to the set s of the most recent record in g.

3. *$\mathcal{A}$ halts and outputs a triple (pk', m', σ'). The output of the experiment is 1 if $\mathbf{Ver}(pk', m', \sigma') = 1$, there is a record (pk, s) in g where $pk = pk'$ and[5] $m' \notin s$. Otherwise, the output is 0.*

2FS is said to be ***Unforgeable With Access to the Token*** *if for all adversaries $\mathcal{A}$ such that $\mathcal{A}(1^\kappa)$ makes at most $q(\kappa)$ queries to the $\mathbf{T}$ oracle after it outputs PWD, and outputs distributions PWD with min entropy lower bounded by $m(\kappa)$, there exist a negligible function μ such that for all κ*

$$\Pr[\mathbf{ExpForgePwdGuess}_{\mathcal{A}}^{2FS}(1^\kappa) = 1] \leq \frac{q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa)$$

Unforgeability for the Token Manufacturer The next definition, *Unforgeability for the Token Manufacturer*, is formalized as an experiment where the adversary first outputs a stateful program Π, and then

[5] Note that this restriction allows $\mathcal{A}$ to ask $\mathbf{T}$ a **Sign** query on input m', as those queries do not modify the list g or any of the sets s.

can ask an honest client (simulated by the challenger) to interact with such program in arbitrary **KeyGen**, **PK** and **Sign** queries (where the adversary can pick the *pwd* and m inputs for such client and receives its outputs). The definition requires that (except with negligible probability) the adversary cannot produce a forgery on a message m valid w.r.t. one of the public keys *pk* output by the client, unless it previously received a valid signature on m w.r.t. *pk* as the output of a **Sign** query.

We restrict such definition to adversaries which satisfy a *compliance* property. Informally, an adversary is compliant if during any execution of the unforgeability experiment, with overwhelming probability, it outputs programs Π such that the outputs of the honest client (simulated by the challenger) on the adversary's queries respect the same correctness conditions as if the simulated client was interacting with an honestly implemented token. In particular, running a **PK** query on input some password *pwd*, the client should obtain the same *pk* which it output during the most recent **KeyGen** query on input the same *pwd*; similarly, the output of a **Sign** query on input m and *pwd* should be a valid signature w.r.t. the public key *pk* which was output during the most recent **KeyGen** query for *pwd*.

Remark 14. Restricting to compliant adversaries is a reasonable limitation: if a user notices that her hardware token is not producing signatures or public keys correctly, for example by selectively aborting during signature generation or by returning invalid signatures or inconsistent public keys, such abnormal behavior would be easy to detect or even impossible to go unnoticed. For example, if a 2FS was used to sign a cryptocurrency transaction, but the client output an invalid signature for the user's expected public key/source address of the transaction, then even if the client side software did not check the signature and it got broadcasted to the network, the receiver of the funds would eventually complain that the funds were never transferred.

Definition 15. *Let* $\mathit{2FS} = (\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$ *be a Two Factor Signature scheme. Consider the following experiment between a PPT adversary* $\mathcal{A}$ *and a challenger, parameterized by a bit* b*:*
$\mathbf{ExpForgeTokMan}_{\mathcal{A}}^{\mathit{2FS}}(1^\kappa)$:

1. $\mathcal{A}(1^\kappa)$ *outputs a polynomial size circuit* Π*, which implements the same interface as a Token Oracle. We stress that this program is not bound to implement the honest algorithms, but may deviate in arbitrary ways (subject to* $\mathcal{A}$ *being compliant as specified below).*

2. $\mathcal{A}$ *can now ask an arbitrary number of* **KeyGen***,* **PK** *and* **Sign** *queries to the challenger. In each query, the challenger simulates an honest client* C *interacting with* Π *in the role of* T *on input a pwd and possibly a message* m *both arbitrarily chosen by the adversary (in the case of a* **Sign** *query,* Π *is also given as input* m*), and gives* $\mathcal{A}$ *such client's output.*

   In addition, for each **KeyGen** *query, the challenger records the simulated client's output pk in an (initially empty) list* g*, and for each* **Sign** *query on input some message* m *where the client's output is* σ*, the challenger adds a record* (pk, m) *to an (initially empty) list* s *for any* $pk \in g$ *such that* $\mathbf{Ver}(pk, m, \sigma) = 1$ *(if such a pk exists).*

3. $\mathcal{A}$ *halts and outputs a triple* (pk', m', σ')*. The output of the experiment is 1 if* $\mathbf{Ver}(pk', m', \sigma') = 1$*,* $pk' \in g$ *and* $(pk', m') \notin s$*. Otherwise, the output is 0.*

During an execution of $\mathbf{ExpForgeTokMan}^{\mathit{2FS}}$*, we say that a query asked by* $\mathcal{A}$ *(i.e. an execution of either* **KeyGen***,***PK** *or* **Sign** *where the challenger executes the algorithm for* C *interacting with* Π *in the role of* T*) is* compliant *if the output of the challenger in this interaction satisfies the same correctness conditions that interacting with an honest token implementation would. In more detail, the query is compliant (with respect to a specific execution of* **ExpForgeTokMan***) if:*

- *in the case of a* **KeyGen** *query, the output of the client (simulated by the challenger) is a* $pk \neq \perp$ *(which implies that* Π *did not abort or send an otherwise invalid message)*

- *in the case of a* **PK** *query on input some password pwd, the simulated client output the same pk which it output the most recent time it executed a* **KeyGen** *query on input the same pwd (or* $\perp$ *if the adversary never asked any* **KeyGen** *query on input pwd)*

- *in the case of a* **Sign** *query on input m and pwd, the simulated client outputs a valid signature w.r.t. the pk which was output during the most recently executed* **KeyGen** *query on input pwd (or $\perp$ if the adversary never asked any* **KeyGen** *query on input pwd).*

We say that an execution of $\mathbf{ExpForgeTokMan}^{\mathit{2FS}}$ *is compliant if all the queries in that execution are compliant. We say that an adversary $\mathcal{A}$ is compliant if, with all but negligible probability, any execution of* $\mathbf{ExpForgeTokMan}_{\mathcal{A}}^{\mathit{2FS}}(1^\kappa)$ *is compliant.*

2FS is said to be ***Unforgeable for the Token Manufacturer*** *if for all PPT compliant adversaries $\mathcal{A}$ there exist a negligible function μ such that for all κ*

$$\Pr[\mathbf{ExpForgeTokMan}_{\mathcal{A}}^{\mathit{2FS}}(1^\kappa) = 1] < \mu(\kappa)$$

For simplicity, we say that a 2FS is unforgeable if it satisfies all the notions of unforgeability.

Definition 16. *We say that a Two Factor Signature scheme is* ***Unforgeable*** *if it is Unforgeable for the Client, Unforgeable for the Token, Unforgeable for the Token Manufacturer and Unforgeable with Access to the Token.*

Non-Signalling Towards proving unforgeability for the token manufacturer, it will be useful to first show that our scheme satisfies a notion of *Non-Signalling*, which is of independent interest. This property is formalized as an indistinguishability definition: the adversary outputs a circuit Π, and then asks the challenger to interact with such circuit on arbitrary **KeyGen**, **PK** and **Sign** queries. The challenger either uses Π to answer all such queries, or an honest implementation of the token algorithms; we require that no adversary can notice this difference with better than negligible probability. As in the previous definition, we restrict our attention to *compliant* adversaries.

Definition 17. *Let* $\mathit{2FS} = (\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$ *be a Two Factor Signature scheme. Consider the following experiment between an adversary $\mathcal{A}$ and a challenger, parameterized by a bit b:*
$\mathbf{ExpNonSignal}_{\mathcal{A}}^{\mathit{2FS},b}(1^\kappa)$:

1. *$\mathcal{A}(1^\kappa)$ outputs a polynomial sized circuit Π, which implements the same interface as a Token Oracle. We stress that this program is not bound to implement the honest algorithms, but may deviate in arbitrary ways (subject to $\mathcal{A}$ being compliant as specified below).*

2. *$\mathcal{A}$ can now ask an arbitrary number of* **KeyGen**, **PK** *and* **Sign** *queries to the challenger. In each query, the adversary provides the inputs for C (i.e. pwd and possibly m). If $b = 0$, the challenger interacts with program Π using the appropriate algorithms for C and the inputs given by $\mathcal{A}$ (note that in the case of a* **Sign** *query, Π is also given the message m supplied by the adversary as an input), and gives $\mathcal{A}$ the local output of the C algorithm in such computation. If $b = 1$, instead, the challenger answers the queries by interacting with an honestly implemented Token Oracle.*

3. *$\mathcal{A}$ halts and outputs a bit b', which defines the output of the experiment.*

Note that in an execution of $\mathbf{ExpNonSignal}^{\mathit{2FS},0}$*, $\mathcal{A}$'s view has exactly the same distribution as in an execution of* **ExpForgeTokMan***. Thus, we can define a compliant query asked by $\mathcal{A}$ w.r.t. an* $\mathbf{ExpNonSignal}^{\mathit{2FS},0}$ *execution, a compliant execution of* $\mathbf{ExpNonSignal}^{\mathit{2FS},0}$ *and a compliant adversary as in Definition 15.*

2FS is said to be ***Non-Signalling*** *if for all compliant PPT adversaries $\mathcal{A}$ there exist a negligible function μ such that for all κ*

$$|\Pr[\mathbf{ExpNonSignal}_{\mathcal{A}}^{\mathit{2FS},0}(1^\kappa) = 1] - \Pr[\mathbf{ExpNonSignal}_{\mathcal{A}}^{\mathit{2FS},1}(1^\kappa) = 1]| < \mu(\kappa)$$

5 Constructing a Two Factor Signature Scheme

In this section, we show how to construct a secure Two Factor Signature scheme (in the random oracle model), by combining any IND-CPA and INT-CTXT secure Symmetric Encryption scheme, a hash function (modelled as a random oracle) and any Unforgeable and Statistically Non-Signalling Threshold Signature scheme.

Let $\mathtt{TS} = (\mathbf{TS.GC}, \mathbf{TS.GT}, \mathbf{TS.SC}, \mathbf{TS.ST}, \mathbf{TS.Ver})$ be a Threshold Signature scheme, $\mathtt{SE} = (\mathbf{SE.G}, \mathbf{SE.E}, \mathbf{SE.D})$ be a Symmetric Encryption scheme, and $\mathbf{RO}_\kappa$ be hash function which maps strings of arbitrary length to $\{0,1\}^\kappa \times \{0,1\}^\kappa$. Our proposed construction depends on a security parameter κ, which is given as implicit input to all algorithms.

The token state s_T is structured as a key-value store (map), where the keys are strings in $\{0,1\}^\kappa$ called *handles* and the values are tuples of strings. Initially, the $\mathbf{KeyGen_T}$ algorithm can be supplied $\perp$, which is treated as an empty store. We define $s_T.\mathbf{Add}(\mathit{handle}, y)$ as the map obtained from s_T by adding the key-value pair (handle, y) (which overwrites any previous value associated with handle), and $s_T.\mathbf{Find}(\mathit{handle})$ as the value associated to *handle* by s_T, or $\perp$ if no such pair exists.

All algorithms will abort (i.e. return $\perp$) if any of their sub-algorithms abort (for example if decrypting a ciphertext fails or the store s_T does not contain the expected value) or the other party aborts or sends a malformed message. Using these conventions, we can define a Two Factor Signature scheme as follows (the scheme is also illustrated in Fig. 1):

- $\mathbf{KeyGen_C}(\mathit{pwd}) \to \mathit{pk}$: Run $\mathbf{TS.GC}(1^\kappa)$ interacting with $\mathbf{KeyGen_T}$ and obtain $(\mathit{sk}_C, \mathit{pk})$ as the local output. Then, compute $(\mathit{ek}, \mathit{handle}) \leftarrow \mathbf{RO}_\kappa(\mathit{pwd}), c \leftarrow \mathbf{SE.E}(\mathit{ek}, (\mathit{sk}_C, \mathit{pk}))$ and send (handle, c) to T. Output pk.

- $\mathbf{KeyGen_T}(s_T) \to s'_T$: Run $\mathbf{TS.GT}(1^\kappa)$ interacting with $\mathbf{KeyGen_C}$ and obtain $\mathit{sk}_T, \mathit{pk}$ as the local output. Then, receive (handle, c) from $\mathbf{KeyGen_C}$, set $s'_T \leftarrow s_T.\mathbf{Add}(\mathit{handle}, (c, \mathit{sk}_T, \mathit{pk}))$ and output (s'_T, pk).

- $\mathbf{PK_C}(\mathit{pwd})$: Compute $(\mathit{ek}, \mathit{handle}) \leftarrow \mathbf{RO}_\kappa(\mathit{pwd})$, send *handle* to $\mathbf{PK_T}$. Upon receiving c in response, compute $(\mathit{sk}_C, \mathit{pk}) \leftarrow \mathbf{SE.D}(\mathit{ek}, c)$ and output pk.

- $\mathbf{PK_T}(s_T)$: Upon receiving *handle* from $\mathbf{PK_C}$, retrieve from the state $(c, \mathit{sk}_C, \mathit{pk}) \leftarrow s_T.\mathbf{Find}(\mathit{handle})$, send c to $\mathbf{PK_C}$ and output pk.

- $\mathbf{Sign_C}(\mathit{pwd}, m)$: Compute $(\mathit{ek}, \mathit{handle}) \leftarrow \mathbf{RO}_\kappa(\mathit{pwd})$ and send *handle* to $\mathbf{Sign_T}$. Upon receiving c in response, compute $(\mathit{sk}_C, \mathit{pk}) \leftarrow \mathbf{SE.D}(\mathit{ek}, c)$, then execute $\mathbf{TS.SC}(\mathit{sk}_C, m)$ (interacting with $\mathbf{Sign_T}$) and output the resulting σ.

- $\mathbf{Sign_T}(s_T, m)$: Upon receiving *handle* from $\mathbf{PK_C}$, compute $(c, \mathit{sk}_C, \mathit{pk}) \leftarrow s_T.\mathbf{Find}(\mathit{handle})$, send c to $\mathbf{Sign_C}$ and run $\mathbf{TS.ST}(\mathit{sk}_T, m)$.

- $\mathbf{Ver}(\mathit{pk}, m, \sigma)$: Output $\mathbf{TS.Ver}(\mathit{pk}, m, \sigma)$.

The security of the scheme is established by the following theorems. We provide proof sketches here, and defer the full proofs to Appendix A.

Theorem 18. *If the underlying Threshold Signature scheme is Unforgeable for the Client, the Two Factor Signature scheme described above is Unforgeable for the Client.*

Proof Sketch. This is essentially a reduction to the unforgeability for C of the Threshold Signature scheme. The adversary $\mathcal{B}$ (against the TS) simulates for any adversary $\mathcal{A}$ (against the 2FS) an execution of **ExpForgeC**; $\mathcal{B}$ guesses which of the **KeyGen** queries by $\mathcal{A}$ will produce a public key pk such that $\mathcal{A}$ outputs a forgery on m w.r.t. pk but $\mathcal{A}$ does not ask any **Sign** queries "with respect to pk" (in a sense explained in the full proof). $\mathcal{B}$ makes $\mathcal{A}$ interact with its challenger for such **KeyGen** query (and the related **Sign** queries), so that if its guess is correct then the forgery produced by $\mathcal{A}$ can directly be used as a forgery to win **TS.ForgeC**. □

$\mathbf{KeyGen_C}(pwd)$:		$\mathbf{KeyGen_T}(s_T)$:
$(sk_C, pk) \leftarrow \mathbf{TS}.\mathbf{GC}(1^\kappa)$	$\leftrightarrow$	$\mathbf{TS}.\mathbf{GT}(1^\kappa) \rightarrow (sk_T, pk)$
$(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$		
$c \leftarrow \mathbf{SE}.\mathbf{E}(ek, (sk_C, pk))$	$\xrightarrow{handle,c}$	$s'_T \leftarrow s_T.\mathbf{Add}(handle, (c, sk_T, pk))$
Output pk		Output (s'_T, pk)
$\mathbf{PK_C}(pwd)$:		$\mathbf{PK_T}(s_T)$:
$(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$	$\xrightarrow{handle}$	$(c, sk_T, pk) \leftarrow s_T.\mathbf{Find}(handle)$
$(sk_C, pk) \leftarrow \mathbf{SE}.\mathbf{D}(ek, c)$	$\xleftarrow{c}$	
Output pk		Output pk
$\mathbf{Sign_C}(pwd, m)$:		$\mathbf{Sign_T}(s_T, m)$:
$(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$	$\xrightarrow{handle}$	$(c, sk_T, pk) \leftarrow s_T.\mathbf{Find}(handle)$
$(sk_C, pk) \leftarrow \mathbf{SE}.\mathbf{D}(ek, c)$	$\xleftarrow{c}$	
$\sigma \leftarrow \mathbf{TS}.\mathbf{SC}(sk_C, m)$	$\leftrightarrow$	$\mathbf{TS}.\mathbf{ST}(sk_T, m)$
Output σ		

Figure 1: The Two Factor Signature scheme construction. The verification algorithm is the one of the underlyihg TS.

Theorem 19. *If* TS *is Unforgeable for the Token, and* SE *is both IND-CCA and INT-CTXT secure, the Two Factor Signature scheme described above is Unforgeable for the Token.*

Proof Sketch. The proof is structured as an hybrid argument. In the first hybrid, we abort if the adversary makes a random oracle query for the password picked by the challenger. If this does not happen, then the secret encryption key used by the simulated client when interacting with the adversary is hidden from its view, and therefore in the following hybrids we can abort if the adversary tries to manipulate such ciphertexts (thanks to the INT-CTXT security of the encryption scheme) and moreover substitute all the ciphertexts with encryptions of 0 (thanks to the IND-CCA security). At this point, the threshold signing key shares used by the client are hidden from the adversary and therefore, as in the proof of unforgeability for the client, the challenger can reduce forgeries for the Two Factor Signature scheme to forgeries for the Threshold Signature scheme. □

Theorem 20. *If the underlying Threshold Signature scheme is Unforgeable for the Client, the Two Factor Signature scheme described above is Unforgeable with Access to the Token.*

Proof Sketch The proof is structured as an hybrid argument. First, we abort if the adversary interacts with the token using a *handle* derived as a hash of the password sampled by the challenger (which probability can be bounded by $\frac{q(\kappa)}{2^{m(\kappa)}}$). Then, since the adversary cannot query the token using the correct password, we can reduce to the Unforgeability for the Client property of the Threshold Signature scheme similarly to the proof of Thereom 18. □

Theorem 21. *Assuming the underlying Threshold Signature scheme is Statistically Non-Signalling, the Two Factor Signature scheme described above is Non-Signalling.*

Proof Sketch. The proof is structured as an hybrid argument on the number of queries made by the adversary. Starting from the experiment where the challenger always uses the circuit Π output by the adversary to answer all queries, we progressively substitute such answers one at a time, starting from the last query. Signing queries on a message m which should be produced w.r.t. a public key that the adversary has already seen are substituted with a randomly sampled signature on m with respect to the same public key, while queries for new public keys are answered by running $(sk_C, sk_T, pk) \leftarrow \mathbf{TS}.\mathbf{Gen}(1^\kappa)$ (i.e. by running the threshold key generation algorithm honestly and without interacting with Π) and returning the resulting pk to $\mathcal{A}$. We prove that an adversary who can distinguish between two adjacent hybrids can contradict one of the two Non-Signalling property of the Threshold Signature scheme. Moreover, in the last hybrid the view of the adversary does not depend on the circuit Π, and so we can switch in an analogous way to an experiment where the challenger always uses an honest token oracle. Note that sampling signatures at random without knowing the corresponding secret key shares makes the reduction require exponential time, but this is not a problem because the Non-Signalling properties of the Threshold Signature scheme hold even against an exponential time adversary. □

Theorem 22. Assuming the underlying Threshold Signature scheme is Statistically Non-Signalling and Unforgeable for the Client, the TFS described above is also Unforgeable for the Token Manufacturer.

Proof Sketch. The proof is structured as an hybrid argument. First, instead of using the circuit Π output by the adversary, all queries by $\mathcal{A}$ are answered using an honestly implemented token oracle. Due to the Non-Signalling property of the 2FS, this cannot affect $\mathcal{A}$'s view and therefore its success probability. Given that $\mathcal{A}$ is now interacting with an honest token, we can prove that $\mathcal{A}$ cannot forge using a similar argument as in the proof of Unforgeability for the Client. □

6 Constructions of Non-Signalling TS

6.1 A Schnorr-based TFS

In this section, we show that a simple variation of the Schnorr-based threshold scheme presented by Nicolosi et al. [15] satisfies definitions 8 and 9, and can thus be used in our construction to obtain a secure Two Factor Signature scheme. With respect to the original scheme, the only difference is that we require the client to send a nonce to the token (in their paper, the second party is called the "server"), which the token will use in all the "random oracle based" commitments which it produces. This is required because in our Non-Signalling proof, the exponential time adversary $\mathcal{A}$ might otherwise "suggest" to the circuit Π two openings for the same commitment, which would allow Π to bias the distribution of the outputs of the protocol.

We summarize the modified protocol in figure 2 and refer the reader to [15] for more discussion.

This scheme (as well as the ECDSA based one presented in the next section) is parameterized by a group $\mathbb{G}$ of order q with generator g, where $q > 2^\kappa$, and two hash functions H, G whose range is $\{0,1\}^\kappa$ (treated as random oracles). More formally, to define an asymptotically secure scheme, where the scheme only depends on the security parameter κ, we assume the existence of a deterministic group generator $\mathcal{G}$ such that $\mathcal{G}(1^k)$ samples $\mathbb{G}, q, g$ as above. Picking primes and generators typically requires random coins. We here require these random coins are given non-uniformly to make the group generation algorithm deterministic (in practice, we simply rely on groups picked by e.g., NIST). Technically, this makes all of our cryptographic primitives non-uniform constructions (since $\mathcal{G}$ is now possibly non-uniform), but we ignore this slight mismatch as it has no impact.

Theorem 23. Assuming the discrete logarithm assumption holds in $\mathbb{G}$, the protocol of Figure 2 is Unforgeable per Def. 8 and Non-Signalling per Def. 9 (in the random oracle model).

Proof. First, the proof that the protocol satisfies Definition 8 can be adapted with minor modifications from [15, Theorem 1, Theorem 2]. To satisfy Def. 9, we consider the two experiments $\mathbf{TS}.\mathbf{NS1}$ and $\mathbf{TS}.\mathbf{NS2}$.

For the first one, intuitively, the $\mathbf{TS}.\mathbf{Gen}$ algorithm can be seen as a coin flipping protocol to sample the public key pk uniformly at random, and therefore Π cannot bias the distribution of such output (because it has to commit to its own randomness before seeing the client's one).

More formally, consider the probability $\Pr[\mathbf{TS}.\mathbf{NS1}_{\mathcal{A}}^{\mathtt{2FS},0}(1^\kappa) = 1]$. Expanding the experiment and substituting the steps of $\mathbf{TS}.\mathbf{GC}$, where for notational convenience we consider $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ and $\Pi = (\Pi_1, \Pi_2)$

TS.GC(1^κ) :		**TS.GT**(1^κ) :
$n \leftarrow_R \{0,1\}^\kappa$	$\xrightarrow{n}$	$sk_T \leftarrow_R \mathbb{Z}_q$
	$\xleftarrow{c}$	$c \leftarrow G(n, g^{sk_T})$
$sk_C \leftarrow_R \mathbb{Z}_q$	$\xrightarrow{g^{sk_C}}$	
Check $c = G(n, g^{sk_T})$	$\xleftarrow{g^{sk_T}}$	
$pk \leftarrow g^{sk_T} \cdot g^{sk_C}$		$pk \leftarrow g^{sk_T} \cdot g^{sk_C}$
Output sk_C, pk		Output sk_T, pk
TS.SC(sk_C, m) :		**TS.ST**(sk_T, m) :
$n \leftarrow_R \{0,1\}^\kappa$	$\xrightarrow{n}$	$r_T \leftarrow_R \mathbb{Z}_q$
	$\xleftarrow{c}$	$c \leftarrow G(n, g^{r_T})$
$r_C \leftarrow_R \mathbb{Z}_q$	$\xrightarrow{g^{r_C}}$	$e \leftarrow H(m, g^{r_C} \cdot g^{r_T})$
Check $c = G(n, g^{r_T})$	$\xleftarrow{g^{r_T}, \sigma_T}$	$\sigma_T \leftarrow r_T + e \cdot sk_T$
$R \leftarrow g^{r_C} \cdot g^{r_T}$		
$e \leftarrow H(m, R)$		
$\sigma \leftarrow r_C + e \cdot sk_C + \sigma_T$		
Output (σ, R)		

Figure 2: 2-out-of-2 Threshold Schnorr, adapted from [15].

as each consisting of two separate "stateless" programs which can explicitly pass some state $t_{\mathcal{A}}, t_\Pi$ between each other, we obtain the following experiment:
$\mathsf{Exp}_0(\mathcal{A}, 1^\kappa)$

1. $(\Pi, t_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\kappa)$
2. $n \leftarrow_R \{0,1\}^\kappa$
3. $(c, t_\Pi) \leftarrow \Pi_1(1^\kappa, n)$
4. $sk_C \leftarrow_R \mathbb{G}$
5. $g^{sk_T} \leftarrow \Pi_2(g^{sk_C}; t_\Pi)$
6. Output $\perp$ if $c \neq G(n, g^{sk_T})$
7. $pk \leftarrow g^{sk_C} \cdot g^{sk_T}$
8. Output $\mathcal{A}_2(pk; t_{\mathcal{A}})$

Now consider the following hybrid experiment (changes are in blue):
$\mathsf{Exp}_1(\mathcal{A}, 1^\kappa)$

1. $(\Pi, t_{\mathcal{A}}) \leftarrow \mathcal{A}_1(1^\kappa)$
2. $n \leftarrow_R \{0,1\}^\kappa$
3. $(c, t_\Pi) \leftarrow \Pi_1(1^\kappa, n)$
4. $sk_C \leftarrow_R \mathbb{G}$
5. $g^{sk_T} \leftarrow \Pi_2(g^{sk_C}; t_\Pi)$; $pk' \leftarrow_R \mathbb{G}$; rewind and run $g^{sk'_T} \leftarrow \Pi_2(pk'/g^{sk_T}; t_\Pi)$

6. Output $\perp$ if $c \neq G(n, g^{sk_T})$ or if $c \neq G(n, g^{sk'_T})$
7. $pk \leftarrow g^{sk_C} \cdot g^{sk_T}$
8. Output $\mathcal{A}_2(pk; t_{\mathcal{A}})$

At an high level, in this experiment we are rewinding Π and running a second time on a different input value for g^{sk_C}. Since, by assumption, the probability that Π aborts (or outputs an invalid commitment opening which would cause **TS**.**SC** to abort) is bounded by $\mu(\kappa)$, and in Exp_1 the distributions of g^{sk_C} is identical to the one of pk'/g^{sk_T} (both are uniformly distributed in $\mathbb{G}$), the probabilities associated with these two experiments differ by at most a negligible function in κ.

Next consider another hybrid Exp_2, defined as Exp_1 but where in step 6 we also output $\perp$ if $g^{sk_T} \neq g^{sk'_T}$. We have that even in this case $|\Pr[\mathsf{Exp}_1(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathsf{Exp}_2(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa)$. This is because the difference between the two probabilities can be bounded by the probability that Π finds a collision in the random oracle G while making only a polynomial number of queries. Note that, while $\mathcal{A}$ runs in exponential time and can thus find collisions in G, it can only embed a useful collision into Π with negligible probability: since the collision needs to contain the nonce n which is sampled by the experiment after Π was output, and Π is described in polynomial space, it can only contain collisions for a polynomial number of values of n.

Next consider hybrid Exp_3, defined as Exp_2 except that in the last step the output is determined by $\mathcal{A}_2(pk'; t_{\mathcal{A}})$ instead of $\mathcal{A}_2(pk; t_{\mathcal{A}})$. This change has no effect because the distributions of pk and pk' are identical; steps (4) in the previous experiment and step (5) in this one can be interchanged. However, notice that at this point, the output of Π is no longer in the view of $\mathcal{A}_2$ (as pk' is uniformly distributed in $\mathbb{G}$); in particular, the value of the experiment is identical to $\Pr[\mathbf{TS}.\mathbf{NS1}^{\mathsf{2FS},1}_{\mathcal{A}}(1^\kappa) = 1]$. Thus, overall, we have

$$|\Pr[\mathbf{TS}.\mathbf{NS1}^{\mathsf{2FS},0}_{\mathcal{A}}(1^\kappa) = 1] - \quad \Pr[\mathbf{TS}.\mathbf{NS1}^{\mathsf{2FS},1}_{\mathcal{A}}(1^\kappa) = 1]| < \mu(\kappa)$$

For the second Non-Signalling experiment, consider again any $\mathcal{A}$ that produces a circuit Π and sk_C, m, pk subject to the constraint that $\Pr[\langle\sigma; \cdot\rangle \leftarrow \langle\mathbf{TS}.\mathbf{SC}(sk_C, m); \Pi\rangle : \mathbf{TS}.\mathbf{Ver}(pk, m, \sigma) = 1] > 1 - \mu(\kappa)$.

A very similar argument to the one used in the first property applies. In particular, an analogous sequence of hybrids (with just some variable renaming and an extra output from Π_2 in step 5) can be used to prove that R's distribution is statistically close to uniform. From there, we then argue that the Schnorr signature verification is such that for each public key pk, message m, and group element R, there is exactly one value σ such that (R, σ) verifies under pk for message m. This implies that the overall distribution over signatures is statistically close to a uniform pair that verifies. □

Using the scheme of figure 2 in our generic construction immediately gives the following result.

Theorem 24. *Assuming the discrete logarithm assumption holds in the family of groups $\mathbb{G}$, there exists an unforgeable and Non-Signalling 2FS in the random oracle model that outputs Schnorr signatures.*

6.2 An ECDSA-based TFS

In this section, we present a simple modification of the 2-out-of-2 threshold ECDSA protocol from [5]. This protocol is substantially more complex than the Schnorr case and relies upon the $\mathcal{F}_{\mathsf{Mul}}$ and $\mathcal{F}_{\mathsf{SF\text{-}OT}}$ functionalities which are defined and implemented in [5]. For our purposes, we can treat them as subroutines. With respect to the original scheme from from [5] and as in the Schnorr case, the only difference here is that we instantiate the commitments using a random oracle and require the client to send a nonce to the token (in their paper, the second party is called the "server"), which the token will use as part of these commitments. This modification is requiredto prove the Non-Signalling property.

6.2.1 Setup

We begin with the setup protocol in Fig. 3.

$$\begin{array}{rcl}
\mathbf{TS}.\mathbf{GC}(1^\kappa): & & \mathbf{TS}.\mathbf{GT}(1^\kappa): \\
 & & \\
n \leftarrow_R \{0,1\}^\kappa & \overset{n}{\rightarrow} & sk_T \leftarrow_R \mathbb{Z}_q \\
 & & pk_T \leftarrow g^{sk_T} \\
 & \overset{c}{\leftarrow} & c \leftarrow G(n, pk_T) \\
sk_C \leftarrow_R \mathbb{Z}_q & & \\
pk_{\mathsf{C}} \leftarrow g^{sk_C} & & \\
\pi_{\mathsf{C}} \leftarrow \mathrm{PoK}\{sk_C : pk_{\mathsf{C}}\} & \xrightarrow{pk_C, \pi} & \pi_{\mathsf{T}} \leftarrow \mathrm{PoK}\{sk_T : pk_{\mathsf{T}}\} \\
\text{Check } c = G(n, pk_{\mathsf{T}}) & \xleftarrow{pk_{\mathsf{T}}, \pi_{\mathsf{T}}} & \\
pk \leftarrow pk_{\mathsf{T}}^{sk_C} & & pk \leftarrow pk_{\mathsf{C}}^{sk_T} \\
o_C & \xleftrightarrow{\mathrm{OText}(1^\kappa)} & o_T \\
\text{Output } (sk_C, o_C), pk & & \text{Output } (sk_T, o_T), pk
\end{array}$$

Figure 3: Threshold ECDSA setup protocol. This protocol is parameterized by a group $\mathbb{G}$ of order q with generator g, where $q > 2^\kappa$, and two hash functions H, G whose range is $\{0,1\}^\kappa$ (treated as random oracles).

6.2.2 ECDSA Signing protocol

The signing protocol requires more steps, and we thus express it in longer form. As before, the protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$ and the hash function H.

1. $\mathbf{TS}.\mathbf{ST}(sk_T, m)$: sample instance key, $k_{\mathsf{T}} \leftarrow \mathbb{Z}_q$, compute $D_T \coloneqq g^{k_{\mathsf{T}}}$ and send to client.
2. $\mathbf{TS}.\mathbf{SC}(sk_C, m)$: sample instance key seed, $k'_{\mathsf{C}} \leftarrow \mathbb{Z}_q$ and compute

   $$\begin{aligned} R' &\coloneqq k'_{\mathsf{C}} \cdot D_T \\ k_{\mathsf{C}} &\coloneqq H(R') + k'_{\mathsf{C}} \\ R &\coloneqq k_{\mathsf{C}} \cdot D_T \end{aligned}$$

   choose a pad $\phi \leftarrow \mathbb{Z}_q$
3. $\mathbf{TS}.\mathbf{SC}$ and $\mathbf{TS}.\mathbf{ST}$ invoke the $\mathcal{F}_{\mathsf{Mul}}$ functionality with inputs $\phi + 1/k_{\mathsf{C}}$ and $1/k_{\mathsf{T}}$ respectively, and receive shares t^1_{C} and t^1_{T} of their padded joint inverse instance key

   $$t^1_{\mathsf{C}} + t^1_{\mathsf{T}} = \frac{\phi}{k_{\mathsf{T}}} + \frac{1}{k_{\mathsf{C}} \cdot k_{\mathsf{T}}}$$

   and then invoke $\mathcal{F}_{\mathsf{Mul}}$ with inputs $\mathsf{sk}_{\mathsf{C}}/k_{\mathsf{C}}$ and $\mathsf{sk}_{\mathsf{T}}/k_{\mathsf{T}}$. They receive shares t^2_{C} and t^2_{T} of their joint secret key over their joint instance key

   $$t^2_{\mathsf{C}} + t^2_{\mathsf{T}} = \frac{\mathsf{sk}_{\mathsf{C}} \cdot \mathsf{sk}_{\mathsf{T}}}{k_{\mathsf{C}} \cdot k_{\mathsf{T}}}$$

   The protocol instances that instantiate $\mathcal{F}_{\mathsf{Mul}}$ are interleaved such that the messages from $\mathbf{TS}.\mathbf{ST}$ to $\mathbf{TS}.\mathbf{SC}$ are transmitted first, followed by replies.
4. $\mathbf{TS}.\mathbf{SC}$ transmits R' to $\mathbf{TS}.\mathbf{ST}$, who computes

   $$R \coloneqq H(R') \cdot D_T + R'$$

   For both Alice and Bob let $(r_x, r_y) = R$.
5. $\mathbf{TS}.\mathbf{SC}$ sends a proof of knowledge of k_{C} w.r.t. R and base D_T. $\mathbf{TS}.\mathbf{ST}$ aborts if the proof does not verify.

6. **TS.SC** and **TS.ST** both compute $m' = H(m)$.
7. **TS.SC** computes the first check value $\mathit{\Gamma}^1$, encrypts pad ϕ with $\mathit{\Gamma}^1$, and transmits the encryption η^ϕ to **TS.ST**.
$$\begin{aligned} \mathit{\Gamma}^1 &:= G + \phi \cdot k_{\mathsf{C}} \cdot G - t^1_{\mathsf{C}} \cdot R \\ \eta^\phi &:= H(\mathit{\Gamma}^1) + \phi \end{aligned}$$
8. **TS.SC** computes signature share σ_{C} and the second check value $\mathit{\Gamma}^2$, encrypts σ_{C} with $\mathit{\Gamma}^2$ and then transmits the encryption η^σ to **TS.ST**
$$\begin{aligned} \sigma_{\mathsf{C}} &:= (m' \cdot t^1_{\mathsf{C}}) + (r_x \cdot t^2_{\mathsf{C}}) \\ \mathit{\Gamma}^2 &:= (t^1_{\mathsf{C}} \cdot pk) - (t^2_{\mathsf{C}} \cdot G) \\ \eta^\sigma &:= H(\mathit{\Gamma}^2) + \sigma_{\mathsf{C}} \end{aligned}$$
9. **TS.ST** computes the check values and reconstructs the signature
$$\begin{aligned} \mathit{\Gamma}^1 &:= t^1_{\mathsf{T}} \cdot R \\ \phi &:= \eta^\phi - H(\mathit{\Gamma}^1) \\ \theta &:= t^1_{\mathsf{T}} - \phi / k_{\mathsf{T}} \\ \sigma_{\mathsf{T}} &:= (m' \cdot \theta) + (r_x \cdot t^2_{\mathsf{T}}) \\ \mathit{\Gamma}^2 &:= (t^2_{\mathsf{T}} \cdot G) - (\theta \cdot pk) \\ \sigma &:= \sigma_{\mathsf{T}} + \eta^\sigma - H(\mathit{\Gamma}^2) \end{aligned}$$
10. **TS.ST** uses the public key pk to verify that (r_x, σ) is a valid signature on message m. If the verification fails, **TS.ST** aborts. If it succeeds, output the pair (r_x, σ') where σ' is the value in $\{\sigma, -\sigma\}$ that is less than $q/2$.
11. **TS.SC** verifies the pair (r_x, σ) on message m and pk and verifies that $\sigma < q/2$.

Theorem 25. *Assuming the CDH Assumption for the family of group parameters $\mathbb{G}$ and that the ECDSA signature is a secure signature scheme for the family $\mathbb{G}$ in the random oracle model, the threshold signature scheme* (**TS.GC**, **TS.GT**, **TS.SC**, **TS.ST**) *described above satisfies Definition 8 and Definition 9.*

Proof Sketch. The unforgeability property (Definition 8) follows from [5, Theorem E.1] under the same assumptions. For the first Non-Signalling property, the key observation is that the first three messages are the same as the Schnorr setup protocol in Fig. 2 with the extra requirement here that a proof of knowledge is given for the secret shares. The additional OT extension step in setup does not change the public key pk, but merely add elements to sk_C. It therefore follows for essentially the same argument as for Theorem 23 that

$$|\Pr[\mathbf{TS.NS1}^{\mathsf{2FS},0}_{\mathcal{A}}(1^\kappa) = 1] - \quad \Pr[\mathbf{TS.NS1}^{\mathsf{2FS},1}_{\mathcal{A}}(1^\kappa) = 1]| < \mu(\kappa)$$

For the second Non-Signalling property, observe that **TS.ST** no longer commits to its instance key in the first message; rather, the value is sent in the clear to **TS.SC** in the first message. As discussed in [5], the 2-round protocol for threshold ECDSA allows **TS.SC** to bias the signature, but does not enable **TS.ST** to do so. This is not a problem for our application because Non-Signalling is only required of the token.

Specifically, it follows by inspection that **TS.SC** chooses a random R' in step (2), and therefore by the random oracle, k_{C} is also uniformly distributed in $\mathbb{Z}_q$, and thus R is also uniformly distributed in $\mathbb{G}$. The rest of the steps in the protocol ensure that σ is computed correctly and that no information about shares of the secret key is leaked; however, these steps do not change the distribution of the signature itself. The signature output consists of a pair $(r_x, \sigma) \in \mathbb{Z}_q{}^2$ that passes the verification equality where r_x is the x-coordinate of group element R. The verification for the message m and public key pk is performed by ensuring that both r_x, σ are in the appropriate range (i.e., that $r_x \in [1, q]$ and that $\sigma \in [0, q/2]$), and then calculating

$$(r'_x, r'_y) = R' := g^{H(m)/\sigma} pk^{r_x/\sigma}$$

and checking if $(r'_x \mod q) = (r_x \mod q)$. For each r_x there are exactly two values $s, -s$ that satisfy this relationship. In the penultimate step, **TS**.**ST** picks the value among those two that is less than $q/2$ which **TS**.**SC** verifies; thus for each r_x, there is only 1 value σ such that (r_x, σ) verifies and **TS**.**SC** does not abort the protocol. Thus for the same reason as for Theorem 23, it also follows that

$$|\Pr[\mathbf{TS}.\mathbf{NS2}_{\mathcal{A}}^{\mathsf{2FS},0}(1^\kappa) = 1] - \Pr[\mathbf{TS}.\mathbf{NS2}_{\mathcal{A}}^{\mathsf{2FS},1}(1^\kappa) = 1]| < \mu(\kappa)$$

which establishes that this protocol satisfies Def. 9. □

7 The Unlinkability property

In this section, we define an additional privacy property for a Two Factor Signature scheme, which we call *Unlinkability.* As anticipated in the introduction, this property mandates that an adversary cannot distinguish different public keys generated by the same honest token or the same passwords from public keys generated using different tokens and independently sampled passwords. The same holds even if the adversary is given signatures on arbitrary messages (according to such public keys) and even when the adversary can query such tokens arbitrarily (as long as he does not guess the passwords). This property guarantees that a token, if lost, does not reveal which public keys it can sign for, and that an outside adversary cannot link to each other different cryptocurrency addresses controlled with the same token.

The Unlinkability property is formalized as an experiment parameterized by two bits b_P, b_T (not both 0), where an adversary $\mathcal{A}$ can interact with two token oracles $\mathbf{T}_0$, $\mathbf{T}_1$. $\mathcal{A}$ can first interact with both oracles arbitrarily, and then outputs a distribution on pairs of distinct strings from which the challenger samples a pair of passwords pwd_0, pwd_1. Then, the challengers runs the **KeyGen** algorithm once interacting with $\mathbf{T}_0$ on input pwd_0, and once with $\mathbf{T}_{b_T}$ on input pwd_{b_P} to obtain two public keys pk_α, pk_β. $\mathcal{A}$ is given the two public keys and can keep interacting with the two $\mathbf{T}$ oracles arbitrarily, as well as ask the challenger for signatures on arbitrary messages w.r.t. the two public keys (which the challenger computes by interacting with the appropriate oracle on input the appropriate password). The definition requires that $\mathcal{A}$ should not be able to distinguish between any two different pairs of bits with probability better than the one of guessing one of the two passwords (given one guess per interaction with a token oracle).

For example, in an execution of the experiment where $b_P = 1, b_T = 0$, the two public keys the adversary is given are generated using the same token oracle ($\mathbf{T}_0$), while if $b_P = 1, b_T = 1$ the two public keys would be generated using different tokens: if an adversary cannot distinguish this two cases then the public keys must be "independent" from the token who generated them. Note that if the adversary could guess one of the passwords, it could simply interact with one of the tokens in a **PK** query on input such password and check whether the resulting public key matched pk_α or pk_β, and this attack is unavoidable.

Also, we exclude the case where $b_P = 0, b_T = 0$ as in this case the two public keys given to the adversary would be generated by the same token using the same password, and in our formalization an honest token only associates a single public key to each password at a time (calling **KeyGen** a second time on the first password would associate a new public key with that password and the token will not sign with respect to the old public key any more).

Definition 26. *Let 2FS* = $(\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$ *be a Two Factor Signature scheme. Consider the following experiment between a stateful adversary* $\mathcal{A}$ *and a challenger, parameterized by two bits* b_P, b_T*:*
$\mathbf{ExpUnlink}_{\mathcal{A}}^{\mathit{2FS}, b_P, b_T}(1^\kappa)$:

1. *The challenger runs* $\mathcal{A}$ *with access to two token oracles* $\mathbf{T}^0, \mathbf{T}^1$ *(*$\mathcal{A}$ *is also given the pk values output by such oracles during* **KeyGen** *and* **PK** *queries).*

2. *During the execution, $\mathcal{A}$ can output a distribution $\texttt{PWDS}$ over distinct pairs of passwords (i.e. over $\{0,1\}^\kappa \times \{0,1\}^\kappa$ such that for all $x \in \{0,1\}^\kappa$, the probability of sampling (x,x) is 0). The challenger samples $(pwd_0, pwd_1) \leftarrow \texttt{PWDS}$. It first computes pk_α by interacting in a* **KeyGen** *query with $\mathbf{T}^0$ on input pwd_0, and then computes pk_β by interacting with $\mathbf{T}^{b_T}$ on input pwd_{b_P}. It gives both public keys to $\mathcal{A}$ (we stress that $\mathcal{A}$ is not involved at all in those interactions, besides receiving the challenger's output).*

3. *From this point on, in addition to $\mathbf{T}^0$ and $\mathbf{T}^1$, $\mathcal{A}$ has access to two new oracles O^α and O^β. These oracles accept only* **Sign** *queries, where the adversary provides a message m: for the first oracle, the challenger computes the output σ by executing $\mathbf{Sign_C}(pwd_0, m)$ interacting with $\mathbf{T}^0$; for the second oracle, the challenger computes σ by executing $\mathbf{Sign_C}(pwd_{b_P}, m)$ interacting with $\mathbf{T}^{b_T}$. In both cases σ is returned to $\mathcal{A}$.*

4. *$\mathcal{A}$ halts and outputs a bit, which defines the output of the experiment.*

For each distribution $\texttt{PWDS}$ over $\{0,1\}^\kappa \times \{0,1\}^\kappa$, we can consider the two "truncated" distributions over $\{0,1\}^\kappa$ obtained by sampling an element from $\texttt{PWDS}$ and truncating the first or the last κ bits.

Π is said to be ***Unlinkable*** *if for all PPT $\mathcal{A}$ such that $\mathcal{A}(1^\kappa)$ makes at most $q(\kappa)$ queries to the $\mathbf{T}^0$ and $\mathbf{T}^1$ oracles* after *step 2, and outputs distributions $\texttt{PWDS}$ where each of the truncations has entropy lower bounded by $m(\kappa)$, and for all $b_P, b_T, b'_P, b'_T \in \{0,1\}^4$ such that $(b_P, b_T) \neq (0,0), (b'_P, b'_T) \neq (0,0)$, there exist a negligible function μ such that for all κ*

$$|\Pr[\mathbf{ExpUnlink}_{\mathcal{A}}^{2FS,b_0,b_1}(1^\kappa) = 1] - \Pr[\mathbf{ExpUnlink}_{\mathcal{A}}^{2FS,b_2,b_3}(1^\kappa) = 1]| \leq \frac{4q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa)$$

The following theorem proves that the Two Factor Signature scheme construction presented in section 5 satisfies the Unlinkability definition.

Theorem 27. Assuming $\mathbf{RO}_\kappa$ is a random oracle, the Two Factor Signature scheme presented in section 5 is Unlinkable according to Definition 26.

Proof Sketch. The proof is structured as an hybrid argument. Starting from an execution of $\mathbf{ExpUnlink}^{\Pi,b_P,b_T}$, we can introduce a first hybrid where the experiment is aborted if the adversary sends to any of the tokens a handle derived as a hash of one of the two passwords sampled by the challenger. The probability of $\mathcal{A}$ noticing this difference can be bounded by $\frac{2q(\kappa)}{2^{m(\kappa)}}$ (which bounds the probability that $\mathcal{A}$ can guess one of the passwords) through a reduction to Lemma 6. Then, we notice that in this hybrid the view of the adversary does not depend on any of the two bits b_P, b_T, and so we can switch to an hybrid where we use b'_P, b'_T instead, and finally to an hybrid where we remove the abort condition (the adversary also has at most an at most $\frac{2q(\kappa)}{2^{m(\kappa)}}$ chance of distinguishing here). The last hybrid is a standard execution of $\mathbf{ExpUnlink}^{\Pi,b'_P,b'_T}$, so the advantage of the adversary can be bounded by $2 \cdot \frac{2q(\kappa)}{2^{m(\kappa)}}$ (plus a negligible amount due to possible collisions in the random oracle). Appendix A contains a more detailed proof. □

References

[1] Jesús F Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In *Eurocrypt*, volume 4004, pages 593–611. Springer, 2006.

[2] Dan Boneh, Xuhua Ding, Gene Tsudik, and Chi-Ming Wong. A method for fast revocation of public key certificates and security capabilities. In *USENIX Security Symposium*, pages 22–22, 2001.

[3] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Virtual smart cards: how to sign with a password and a server, 2016.

[4] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In *Advances in Cryptology – CRYPTO 1989*, pages 307–315. Springer, 1990.

[5] J. Doerner, Y. Kondi, E. Lee, and a. shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 595–612, 2018.

[6] Rosario Gennaro and Steven Goldfeder. Fast multiparty threshold ecdsa with fast trustless setup. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1179–1194. ACM, 2018.

[7] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Robust and efficient sharing of RSA functions. In *Advances in Cryptology – CRYPTO 1996*, pages 157–172. Springer, 1996.

[8] Steven Goldfeder, Rosario Gennaro, Harry Kalodner, Joseph Bonneau, Joshua A Kroll, Edward W Felten, and Arvind Narayanan. Securing bitcoin wallets via a new DSA/ECDSA threshold signature scheme, 2015.

[9] Yehuda Lindell. Fast secure two-party ECDSA signing. In *Advances in Cryptology – CRYPTO 2017*, pages 613–644. Springer, 2017.

[10] Yehuda Lindell and Ariel Nof. Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1837–1854. ACM, 2018.

[11] Philip MacKenzie and Michael K Reiter. Delegation of cryptographic servers for capture-resilient devices. *Distributed Computing*, 16(4):307–327, 2003.

[12] Philip MacKenzie and Michael K Reiter. Networked cryptographic devices resilient to capture. *International Journal of Information Security*, 2(1):1–20, 2003.

[13] Antonio Marcedone, Rafael Pass, and abhi shelat. Minimizing trust in hardware wallets with two factor signatures. Cryptology ePrint Archive, Report 2018/???, 2018.

[14] Microchip. Atecc608a datasheet, 2018.

[15] Antonio Nicolosi, Maxwell N Krohn, Yevgeniy Dodis, and David Mazieres. Proactive two-party signatures for user authentication. In *NDSS*, 2003.

[16] Marek Palatinus, Pavol Rusnak, Aaron Voisine, and Sean Bowe. Mnemonic code for generating deterministic keys (bip39). https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

[17] Tal Rabin. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology – CRYPTO 1998*, pages 89–104. Springer, 1998.

[18] T.C. Sottek. Nsa reportedly intercepting laptops purchased online to install spy malware, December 2013. [Online; posted 29-December-2013; `https://www.theverge.com/2013/12/29/5253226/nsa-cia-fbi-laptop-usb-plant-spy`].

A Deferred proofs

A.1 Proof of Theorem 18

Proof. First, notice that by construction in the proposed 2FS scheme it is not possible that p contains a public key which is not in g. This is because a token oracle $\mathbf{T}$ will output a pk during a $\mathbf{PK}$ query (and thus such pk is part of p) only if such a public key is already part of its internal state, and the only way in which pk could be added to such state is through a $\mathbf{KeyGen_T}$ query, which implies that pk is also part of g. Therefore, $\mathcal{A}$ must be winning the experiment by outputting a message m and more valid forgeries on m than the number of signing queries with $\mathbf{T}$ on input m.

During an execution of the **ExpForgeT** experiment, we say that a forgery (pk_j, σ_j) (produced by the adversary at the end of the experiment) is *associated with* the i-th **KeyGen** query if pk_j is the public key output by the token in the **KeyGen** query. Moreover, we say that a **Sign** query is *associated with* the i-th **KeyGen** query if the *handle* that the adversary sends to the challenger as its first message for the **Sign** query is the same which was used in the last message of the i-th **KeyGen** query, and the i-th **KeyGen** query is the most recent query before the **Sign** query with such property. Intuitively, we only consider the most recent **KeyGen** query because a *handle* is used by the token to retrieve the secret key share it will use on a specific **Sign** query. Asking multiple **KeyGen** queries where the same handle is used results in a new secret key share overwriting the old one.

Given all the above, for any adversary $\mathcal{A}$, if the output of an execution of $\mathbf{ExpForgeC}_{\mathcal{A}}^{\mathbf{2FS}}(1^\kappa)$ is 1, then in that execution there must be at least one index i such that the adversary output a forgery (on a message m) associated with the i-th **KeyGen** query, but there is no **Sign** query for m associated with the i-th **KeyGen** query. This is because no two forgeries can be associated with the same **KeyGen** query (as all the public keys in the forgeries must be distinct), each **Sign** query can be associated with at most one **KeyGen** query, and the number of **Sign** queries for m must be smaller than the number of forgeries. We will call an index i which satisfies this property *important.*

Assume by contradiction that there exists an adversary $\mathcal{A}$ which has non negligible probability of winning the $\mathbf{ExpForgeC}_{\mathcal{A}}^{2FS}$ experiment. We can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ which contradicts the unforgeability for C of the underlying Threshold Signature scheme. Let $d(\kappa)$ be an upper bound on the number of **KeyGen** queries asked by $\mathcal{A}$ to the **T** oracle. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of **ExpForgeC**. First, it samples $i \leftarrow_R \{1, \dots, d(\kappa)\}$, and its success probability will depend on such index i being important for the **ExpForgeC** execution. Then it simulates the **T** oracle for $\mathcal{A}$ as follows.

- All queries (of any kind) before the i-th **KeyGen** query are simulated as the honest challenger would. In particular, $\mathcal{B}$ picks $\perp$ as initial state s_T for the token oracle and handles the queries by executing the honest algorithms on input s_T (which is also updated as a result of **KeyGen** queries).

- The i-th **KeyGen** query by the adversary is handled differently. First, $\mathcal{B}$ makes $\mathcal{A}$ interact with its own challenger in an execution of **TS**.**Gen**, where the challenger runs **TS**.**GT** and $\mathcal{A}$ plays the role of C. At the end, $\mathcal{B}$ is given by its challenger the public key pk^* which is part of its local output (note that $\mathcal{B}$ does not learn the challenger's secret key share). Then, when $\mathcal{A}$ sends to the simulated oracle **T**, as the last message of the **KeyGen** query, a message $(handle^*, c^*)$, $\mathcal{B}$ records such message together with pk^*. It completes the query by giving $\mathcal{A}$ the public key pk^*.

- After the i-th **KeyGen** query, when $\mathcal{A}$ asks a **Sign** query to **T** on input m, where $\mathcal{A}$'s first message is equal to $handle^*$, then $\mathcal{B}$ first sends the recorded c^* to $\mathcal{A}$ as a response from **T**. Then, it continues simulating the query by having $\mathcal{A}$ interact with its own challenger in an execution of **TS**.**Sign**. Analogously, **PK** queries where the first message from $\mathcal{A}$ is equal to $handle^*$ are handled by returning to $\mathcal{A}$ the recorded c^*. All other **Sign** and **PK** queries (where $handle \neq handle^*$) are simulated honestly. **KeyGen** queries are also simulated honestly. In addition, if at the end of one such **KeyGen** query the adversary uses as a handle $handle^*$, then $\mathcal{B}$ continues by simulating all subsequent queries (of any kind) honestly, including the ones for $handle^*$ (in particular, $\mathcal{B}$ does not make any more queries to its own challenger).

- When $\mathcal{A}$ outputs a message m and a list of forgeries $(pk_1, \sigma_1), \dots, (pk_n, \sigma_n)$, then if there exists an index j such that $pk^j = pk^*$ and $\mathbf{Ver}(pk^*, m, \sigma_j) = 1$, then $\mathcal{B}$ outputs (m, σ_j) as a forgery to its challenger; otherwise $\mathcal{B}$ halts without producing any forgery.

First of all, notice that $\mathcal{A}$'s view in this experiment has the same distribution as its view in the $\mathbf{ExpForgeC}_{\mathcal{A}}^{2FS}$ one. This is because the only difference between the two is that in some of the queries by the adversary where $handle = handle^*$, $\mathcal{B}$ provides answers to $\mathcal{A}$ through its challenger (which implements the algorithms honestly with a state consistent with what $\mathcal{A}$ would expect), and thus this difference does not affect $\mathcal{A}$'s view.

Moreover, we have that if the simulated **ExpForgeC** outputs 1 and $\mathcal{B}$ guessed the important index i correctly (which happens with probability $\frac{1}{d(\kappa)}$ conditioned on the former event), then $\mathcal{B}$'s output will also be a valid forgery for **TS**.**ForgeC**.

Thus we have that $\Pr[\mathbf{ExpForgeC}_{\mathcal{A}}^{2FS}(1^\kappa) = 1] \leq d(\kappa) \cdot \Pr[\mathbf{TS}.\mathbf{ForgeC}_{\mathcal{B}}^{TS}(1^\kappa) = 1]$, which contradicts the unforgeability against the client of the Threshold Signature scheme. □

A.2 Proof of Theorem 19

Proof. Let $d(\kappa)$ be an upper bound on the number of **KeyGen** queries asked by $\mathcal{A}$. We define the following hybrids:

- $\mathcal{H}_\mathbf{0}$: Defined as $\mathbf{ExpForgeT}_{\mathcal{A}}^{\mathtt{2FS}}(1^\kappa)$
- $\mathcal{H}_\mathbf{1}$: Defined from $\mathcal{H}_\mathbf{0}$, where in addition the experiment is aborted (with a special output `GUESSED`) if the adversary makes a random oracle query for the password *pwd* sampled by the challenger in step 2.
- $\mathcal{H}_\mathbf{2}$: Defined from $\mathcal{H}_\mathbf{1}$, where in addition the experiment is aborted (with a special output `FAKE_CTXT`) if $\mathcal{A}$ asks a $\mathbf{Sign_C}$ or $\mathbf{PK_C}$ query where it sends to the challenger a ciphertext c such that i) c was not initially given to $\mathcal{A}$ by the challenger as part of a previous $\mathbf{KeyGen_C}$ query and ii) $\mathbf{SE}.\mathbf{D}(ek, c) \neq \perp$ where $(ek, \cdot) \leftarrow \mathbf{RO}_\kappa(pwd)$ and *pwd* is the password chosen by the challenger in step 2.
- $\mathcal{H}_\mathbf{3}$: Defined from $\mathcal{H}_\mathbf{2}$, where for each ciphertext computed by the challenger as $\mathbf{SE}.\mathbf{E}(ek, (sk_C, pk))$ and sent to the adversary in a $\mathbf{KeyGen_C}$ query, the challenger instead computes $c \leftarrow \mathbf{SE}.\mathbf{E}(ek, 0^{2k})$ and stores a record $(c, (sk_C, pk))$. Analogously, during any $\mathbf{PK_C}$ and $\mathbf{Sign_C}$ query, instead of decrypting the ciphertext c sent by the adversary as $\mathbf{SE}.\mathbf{D}(ek, c)$ the challenger uses (sk_C, pk) if it has a record $(c, (sk_C, pk))$, and behaves as if the decryption algorithm returned $\perp$ otherwise.

To prove the theorem, it is sufficient to show that for any PPT $\mathcal{A}$ there exists a negligible function μ such that:

$$|\Pr[\mathcal{H}_\mathbf{0}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = 1]| < \frac{2q(\kappa)}{2^{m(\kappa)}} \tag{1}$$

$$|\Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_\mathbf{2}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{2}$$

$$|\Pr[\mathcal{H}_\mathbf{2}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_\mathbf{3}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{3}$$

$$|\Pr[\mathcal{H}_\mathbf{3}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{4}$$

We will prove that each condition holds separately.

1. Let E be the event "$\mathcal{A}$ makes a random oracle query for the password *pwd* sampled by the challenger in step 2". Note that E has the same probability of happening in both hybrids ($\Pr[\mathcal{H}_\mathbf{0}(\mathcal{A}, 1^\kappa) : E] = \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) : E]$), that conditioned on E not happening the view of $\mathcal{A}$ has exactly the same distribution in both hybrids (in particular, $\Pr[\mathcal{H}_\mathbf{0}(\mathcal{A}, 1^\kappa) = 1|\neg E] = \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = 1|\neg E]$) and that $\Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = \mathtt{GUESSED}] = \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) : E]$. Therefore, the probability that $\mathcal{A}$ can distinguish can be upper bounded by the probability that $\mathcal{A}$ causes $\mathcal{H}_\mathbf{1}$ to output `GUESSED`:

$$\begin{aligned}
&\big|\Pr[\mathcal{H}_\mathbf{0}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = 1]\big| \leq \\
&\quad |\Pr[\mathcal{H}_\mathbf{0}(\mathcal{A}, 1^\kappa) = 1|\neg E] - \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = 1|\neg E]| \cdot \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) : \neg E] + \\
&\quad |\Pr[\mathcal{H}_\mathbf{0}(\mathcal{A}, 1^\kappa) = 1|E] - \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = 1|E]| \cdot \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) : E] \leq \\
&\quad 0 + \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) : E] = \Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = \mathtt{GUESSED}]
\end{aligned} \tag{5}$$

Therefore to prove that equation 1 holds it is enough to show that $\Pr[\mathcal{H}_\mathbf{1}(\mathcal{A}, 1^\kappa) = \mathtt{GUESSED}] < \frac{q(\kappa)}{2^{m(\kappa)}}$. We will prove that this is the case by leveraging Lemma 5. For any PPT adversary $\mathcal{A}$ for $\mathcal{H}_\mathbf{1}$, let's define an adversary $\mathcal{B}$ for **ExpGuess** as follows:

$\mathcal{B}^{\mathbf{Guess}(\cdot)}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$ by simulating for it an execution of $\mathcal{H}_{\mathbf{1}}$. When $\mathcal{A}$ outputs a distribution PWDS, $\mathcal{B}$ forwards such distribution to its challenger. Moreover, the challenger samples $(ek, handle) \leftarrow \{0,1\}^{k} \times \{0,1\}^{\kappa}$, which will be used as the output of the random oracle on input the *pwd* picked by $\mathcal{B}$'s challenger.

- Whenever $\mathcal{A}$ asks a $\mathbf{KeyGen_C}$, $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query, such queries are handled honestly, except that instead of computing $(ek, handle) \leftarrow \mathbf{RO}_{\kappa}(pwd)$, $\mathcal{B}$ uses the values sampled in the previous step.

- $\mathcal{B}$ simulates the random oracle $\mathbf{RO}_{\kappa}$ by sampling answers to $\mathcal{A}$'s queries uniformly at random and storing them in a table so that the same query always receives the same answer. In addition, for each such queries for a string *pwd*, $\mathcal{B}$ submits *pwd* as a guess to its **Guess** oracle: if the response is 1, then $\mathcal{B}$ can halt (and **ExpGuess** outputs 1); otherwise the simulation continues.

- If $\mathcal{A}$ halts (with or without outputting a bit), then $\mathcal{B}$ halts as well (and **ExpGuess** outputs 0).

Notice that until $\mathcal{B}$ halts, the view of $\mathcal{A}$ has the same distribution as in $\mathcal{H}_{\mathbf{1}}$. The only change in $\mathcal{A}$'s view is that, instead of computing $(ek, handle) \leftarrow \mathbf{RO}_{\kappa}(pwd)$, $\mathcal{B}$ samples such values uniformly at random and independently from *pwd*. The only way that $\mathcal{A}$ can notice such difference, is if it asks a random oracle query for *pwd*. But, in this case, $\mathcal{B}$ will halt and win the experiment. This proves that $\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^{\kappa}) = \texttt{GUESSED}] < \Pr[\mathbf{ExpGuess}_{\mathcal{B}}(1^{\kappa}) = 1]$. Moreover, the number of guess that $\mathcal{B}$ makes to its oracle is upper bounded by the number of random oracle queries asked by $\mathcal{A}$, from which by Lemma 5 we have that $\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^{\kappa}) = \texttt{GUESSED}] < \frac{q(\kappa)}{2^{m(\kappa)}}$, which is what we wanted to prove.

2. Analogously as in the previous case, one can show that

$$\left|\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^{\kappa}) = 1] - \Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^{\kappa}) = 1]\right| \leq \\ \Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^{\kappa}) = \texttt{FAKE\_CTXT}]$$

We will prove that $\Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^{\kappa}) = \texttt{FAKE\_CTXT}] < \mu(\kappa)$ by reducing this property to the INT-CTXT security of the Symmetric Encryption scheme. For any PPT adversary $\mathcal{A}$ that distinguishes $\mathcal{H}_{\mathbf{1}}$ from $\mathcal{H}_{\mathbf{2}}$, let's define an adversary $\mathcal{B}$ for **SE.ExpINT-CTXT** as follows:
$\mathcal{B}(1^{\kappa})$:

- $\mathcal{B}$ runs $\mathcal{A}$ by simulating for it an execution of $\mathcal{H}_{\mathbf{2}}$.

- Whenever $\mathcal{A}$ asks a $\mathbf{KeyGen_C}$, $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query, such queries are handled as in $\mathcal{H}_{\mathbf{2}}$, with the following differences.

  In any $\mathbf{KeyGen_C}$ query, instead of directly computing $c \leftarrow \mathbf{SE.E}(ek, (sk_C, pk))$ on its own, $\mathcal{B}$ computes c by asking an encryption query to its own **SE.E** oracle on input (sk_C, pk); moreover, it stores a record $(c, (sk_C, pk))$.

  In any $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ queries, instead of directly computing $(sk_C, pk) \leftarrow \mathbf{SE.D}(ek, c)$ on its own, $\mathcal{B}$ first checks if it has a record $(c, (sk'_C, pk'))$ (for some (sk'_C, pk')). If so, it uses (sk'_C, pk') as the output of such decryption; otherwise, it submits c to its own decryption oracle to compute such output. If the decryption oracle returns a message $m \neq \perp$, $\mathcal{B}$ halts with output FAKE_CTXT (and **SE.ExpINT-CTXT** outputs 1), otherwise the simulation continues (and the $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ queries are aborted as if decryption failed).

  Random oracle queries are handled exactly as in $\mathcal{H}_{\mathbf{1}}$. In particular, if a random oracle query would cause $\mathcal{H}_{\mathbf{1}}$ to output GUESSED, then $\mathcal{B}$ also halts with the same output.

- When $\mathcal{A}$ halts (possibly with some forgeries as output), $\mathcal{B}$ also halts.

Notice that until $\mathcal{B}$ halts, the view of $\mathcal{A}$ has the same distribution as in $\mathcal{H}_{\mathbf{2}}$. This is because the only difference in the two experiments is that in one of them the secret key *ek* used to produce the ciphertexts given by the challenger during any **KeyGen** query is computed as $(ek, \cdot) \leftarrow \mathbf{RO}_{\kappa}(pwd)$ in $\mathcal{H}_{\mathbf{2}}$ and independently sampled by $\mathcal{B}$'s challenger in $\mathbf{SE.ExpINT\text{-}CTXT}^{\texttt{SE}}_{\mathcal{B}}$; however, since both experiments are aborted if $\mathcal{A}$ asks for $\mathbf{RO}_{\kappa}(pwd)$, such change cannot affect its view. Moreover, by construction if during a $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query $\mathcal{A}$ outputs a ciphertext which causes $\mathcal{H}_{\mathbf{3}}$ to output FAKE_CTXT then $\mathcal{B}$ would also output FAKE_CTXT, and the execution of $\mathbf{SE.ExpINT\text{-}CTXT}$ would output 1. This proves that $\Pr[\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa) = \mathtt{FAKE\_CTXT}] \leq \Pr[\mathbf{SE.ExpINT\text{-}CTXT}_{\mathcal{B}}(1^\kappa) = 1]$. Since $\mathtt{SE}$ is INT-CTXT secure, such quantity must be negligible, which is what we wanted to prove.

3. Assume by contradiction that there exists an adversary $\mathcal{A}$ such that $|\Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa) = 1]|$ is non negligible.

We will use $\mathcal{A}$ to build an adversary $\mathcal{B}$ that contradicts the IND-CCA security of $\mathtt{SE}$. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$ by simulating for it an execution of $\mathcal{H}_{\mathbf{2}}$.
- Whenever $\mathcal{A}$ asks a $\mathbf{KeyGen_C}$, $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query, such queries are handled as in $\mathcal{H}_{\mathbf{2}}$, with the following differences.

  In any $\mathbf{KeyGen_C}$ query, instead of directly computing $c \leftarrow \mathbf{SE.E}(ek, (sk_C, pk))$ on its own, $\mathcal{B}$ computes c by asking an encryption query to its own $\mathbf{SE.E}$ oracle on input $((sk_C, pk), 0^{2k})$; moreover, it stores a record $(c, (sk_C, pk))$.

  In any $\mathbf{PK_C}$ or $\mathbf{Sign_C}$, instead of directly computing $(sk_C, pk) \leftarrow \mathbf{SE.D}(ek, c)$ on its own, $\mathcal{B}$ first checks if it has a record $(c, (sk'_C, pk'))$ (for some (sk'_C, pk')) and, if so, uses (sk'_C, pk') as the output of such decryption. Otherwise, it submits c to its own decryption oracle: if such oracle outputs $\perp$, then the simulation continues (even though this specific $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query is aborted as $\mathcal{A}$ submitted an invalid ciphertext); otherwise, $\mathcal{B}$ aborts with output $\mathtt{FAKE\_CTXT}$.

  Random oracle queries are handled exactly as in $\mathcal{H}_{\mathbf{1}}$. In particular, if a random oracle query would cause $\mathcal{H}_{\mathbf{1}}$ to output $\mathtt{GUESSED}$, then $\mathcal{B}$ also halts and outputs $\mathtt{GUESSED}$.
- When $\mathcal{A}$ halts (possibly with some forgeries as output), $\mathcal{B}$ outputs a bit b' computed as in step 4 of $\mathbf{ExpForgeT}$.

Notice that, if $\mathcal{B}$ is running in $\mathbf{SE.ExpIND\text{-}CCA}^{\mathtt{SE}}_{\mathcal{B},0}$ (resp. $\mathbf{SE.ExpIND\text{-}CCA}^{\mathtt{SE}}_{\mathcal{B},1}$), then $\mathcal{A}$'s view has the same distribution as in $\mathcal{H}_{\mathbf{2}}$ (resp. $\mathcal{H}_{\mathbf{3}}$). This is because the only difference in the two experiments is that in one of them the secret key ek used to produce the ciphertexts given by the challenger during any $\mathbf{KeyGen}$ query is computed as $(ek, \cdot) \leftarrow \mathbf{RO}_\kappa(pwd)$ in $\mathcal{H}_{\mathbf{1}}$ and independently sampled by $\mathcal{B}$'s challenger in $\mathbf{SE.ExpIND\text{-}CCA}^{\mathtt{SE}}_{\mathcal{B},0}$; however, since both experiments are aborted if $\mathcal{A}$ asks for $\mathbf{RO}_\kappa(pwd)$, such change cannot affect its view. This proves that $|\Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa) = 1]| \leq |\Pr[\mathbf{SE.ExpIND\text{-}CCA}^{\mathtt{SE}}_{\mathcal{A},0}(1^\kappa) = 1] - \Pr[\mathbf{SE.ExpIND\text{-}CCA}^{\mathtt{SE}}_{\mathcal{A},1}(1^\kappa) = 1]|$, which contradicts the IND-CCA security of $\mathtt{SE}$.

4. First, notice that by construction in an execution of $\mathcal{H}_{\mathbf{3}}$ it is not possible that p contains a public key which is not in g. This is because the $\mathbf{PK_C}$ algorithm will output pk during a $\mathbf{PK}$ query (and thus such pk would be part of p) only after successfully decrypting $(sk_C, pk) \leftarrow \mathbf{SE.D}(ek, c)$, where c is the ciphertext sent by $\mathcal{A}$ as the first message. If such a ciphertext was originally given to $\mathcal{A}$ by the challenger during a $\mathbf{KeyGen_C}$ query, then during such query $\mathbf{KeyGen_C}$ must have output pk, which therefore must be part of g. Instead, if such a ciphertext was not given to $\mathcal{A}$ by the challenger (and still decrypts correctly), then the experiment would be halted with output $\mathtt{FAKE\_CTXT}$. Therefore, whenever $\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa) = 1$, $\mathcal{A}$ must have output a message m and more valid forgeries on m than the number of $\mathbf{Sign_C}$ queries on input m.

During an execution of $\mathcal{H}_{\mathbf{3}}$, we say that a forgery (pk_j, σ_j) (produced by the adversary at the end of the experiment) is *associated with* the i-th $\mathbf{KeyGen_C}$ query if the output of this query is pk_j; moreover, we say that a $\mathbf{Sign}$ query is *associated with* the i-th $\mathbf{KeyGen_C}$ query if the ciphertext c that the adversary sends to the challenger as its first message for the $\mathbf{Sign_C}$ query is the same which $\mathcal{A}$ received as the last message of the i-th $\mathbf{KeyGen_C}$ query. Note that, since with overwhelming probability any two independently generated ciphertexts must be different (regardless of whether they encrypt the same message or not), each $\mathbf{Sign_C}$ query can be associated with at most one $\mathbf{KeyGen_C}$ query[6].

Given all the above, for any adversary $\mathcal{A}$, if the output of an execution of $\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa)$ is 1, then in that execution there must be at least one index i such that the adversary output a forgery (on a message m)

[6] Formally, we are conditioning over this event which happens with overwhelming probability.

associated with the i-th $\textbf{KeyGen}_\mathbf{C}$ query, but there is no $\textbf{Sign}_\mathbf{C}$ query for m associated with the i-th $\textbf{KeyGen}_\mathbf{C}$ query. This is because each forgery must be associated with a different $\textbf{KeyGen}_\mathbf{C}$ query (as all the public keys in the forgeries must be distinct), and the number of $\textbf{Sign}_\mathbf{C}$ queries for m must be smaller than the number of forgeries. We will call an index i which satisfies this property *important*.

Assume by contradiction that there exists an adversary $\mathcal{A}$ such that $\Pr[\mathcal{H}_\mathbf{3}(\mathcal{A}, 1^\kappa) = 1]$ is non negligible. We can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ which contradicts the unforgeability for T of the underlying Threshold Signature scheme. Let $d(\kappa)$ be an upper bound on the number of **KeyGen** queries asked by $\mathcal{A}$. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of **ExpForgeC**. First, it samples $i \leftarrow_R \{1, \ldots, d(\kappa)\}$, and its success probability will depend on such index i being important for this $\mathcal{H}_\mathbf{3}$ execution. When $\mathcal{A}$ outputs a distribution $\texttt{PWD}$, $\mathcal{B}$ samples $pwd \leftarrow \texttt{PWD}$ and $(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$. Then it handles queries asked by $\mathcal{A}$ as follows.

- All queries (of any kind) before the i-th **KeyGen** query are simulated as the honest challenger would in an execution of $\mathcal{H}_\mathbf{3}$.

- The i-th **KeyGen** query by the adversary is handled differently. First, $\mathcal{B}$ makes $\mathcal{A}$ interact with its own challenger in an execution of $\mathbf{TS}.\mathbf{Gen}$, where the challenger runs $\mathbf{TS}.\mathbf{GC}$ and $\mathcal{A}$ plays the role of T. At the end, $\mathcal{B}$ is given by its challenger the public key pk^* which is part of its local output (note that $\mathcal{B}$ does not learn the challenger's secret key share). $\mathcal{B}$ computes $c^* \leftarrow \mathbf{SE}.\mathbf{E}(ek, 0^{2k})$,sends $\mathcal{A}$ the couple $handle, c^*$ and creates a record (c, pk^*).

- After the i-th $\textbf{KeyGen}_\mathbf{C}$ query, when $\mathcal{A}$ asks a $\textbf{Sign}_\mathbf{C}$ query on input m, where $\mathcal{A}$'s first message is c^*, then $\mathcal{B}$ makes $\mathcal{A}$ interact with its own challenger in an execution of $\mathbf{TS}.\mathbf{Sign}$, at the end of which it returns to $\mathcal{A}$ the challenger's local output σ. Analogously, **PK** queries where the first message from $\mathcal{A}$ is c^* are handled by returning to $\mathcal{A}$ the recorded pk^*. All other **Sign** and **PK** queries (where $c \neq c^*$) are simulated as in $\mathcal{H}_\mathbf{3}$.

- When $\mathcal{A}$ outputs a message m and a list of forgeries $(pk_1, \sigma_1), \ldots, (pk_n, \sigma_n)$, then if there exists an index j such that $pk^j = pk^*$ and $\mathbf{Ver}(pk^*, m, \sigma_j) = 1$, then $\mathcal{B}$ outputs (m, σ_j) as a forgery to its challenger; otherwise $\mathcal{B}$ halts without producing any forgery.

Notice that $\mathcal{A}$'s view in this experiment has the same distribution as its view in $\mathcal{H}_\mathbf{3}$. This is because the only difference between the two is that in some of the queries by the adversary where $c = c^*$, $\mathcal{B}$ provides answers to $\mathcal{A}$ through its challenger (which implements the algorithms honestly with a state consistent with what $\mathcal{A}$ would expect), and thus this difference does not affect $\mathcal{A}$'s view. Moreover, we have that if the simulates $\mathcal{H}_\mathbf{3}$ execution outputs 1 and $\mathcal{B}$ guessed the important index i correctly, then $\mathcal{B}$'s output will also be a valid forgery for $\mathbf{TS}.\mathbf{ForgeC}$. Thus we have that $\Pr[\mathcal{H}_\mathbf{3}(\mathcal{A}, 1^\kappa) = 1] \leq \Pr[\mathbf{TS}.\mathbf{ForgeC}_\mathcal{B}^{TS}(1^\kappa) = 1]$, which contradicts the unforgeability against the client of the Threshold Signature scheme. □

A.3 Proof of Theorem 20

Proof. Let $d(\kappa)$ be an upper bound on the number of **KeyGen** queries asked by $\mathcal{A}$. We define the following hybrids:

- $\mathcal{H}_\mathbf{0}$: Defined as $\mathbf{ExpForgePwdGuess}_\mathcal{A}^{\texttt{2FS}}(1^\kappa)$

- $\mathcal{H}_\mathbf{1}$: Defined from $\mathcal{H}_\mathbf{0}$, where in addition the experiment is aborted (with a special output $\texttt{GUESSED}$) if after step 2 the adversary makes a **KeyGen PK** or **Sign** query to **T** where it sends $handle \leftarrow \mathbf{RO}_\kappa(pwd)$ to the challenger (either as part of its last message in **KeyGen** queries or as the first message in $\mathbf{PK}_\mathbf{C}$ and $\textbf{Sign}_\mathbf{C}$ queries) and pwd is the password sampled by the challenger in step 2.

To prove the theorem, it is sufficient to show that for any PPT $\mathcal{A}$ there exists a negligible function μ such that:

$$|\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]| < \frac{q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa) \tag{6}$$

$$|\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{7}$$

We will prove that each condition holds separately.

1. Analogously to the first case of the proof of theorem 19, one can show that (here the security parameter argument is elided):

$$\big|\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}) = 1]\big| \leq \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}) = \texttt{GUESSED}]$$

Therefore to prove that equation 1 holds it is enough to show that $\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] < \frac{q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa)$. We will prove that this is the case by leveraging Lemma 5. For any PPT adversary $\mathcal{A}$ for $\mathcal{H}_{\mathbf{1}}$, let's define an adversary $\mathcal{B}$ for **ExpGuess** as follows:
$\mathcal{B}^{\mathbf{Guess}(\cdot)}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$ by simulating for it an execution of $\mathcal{H}_{\mathbf{1}}$. When $\mathcal{A}$ outputs a distribution PWDS, $\mathcal{B}$ forwards such distribution to its challenger. Moreover, the challenger samples $(ek^*, handle^*) \leftarrow \{0,1\}^k \times \{0,1\}^\kappa$, which will be used as the output of the random oracle on input the *pwd* picked by $\mathcal{B}$'s challenger.

- $\mathcal{B}$ simulates the random oracle $\mathbf{RO}_\kappa$ by sampling answers to $\mathcal{A}$'s queries uniformly at random and storing them in a table so that the same query always receives the same answer.

- Whenever $\mathcal{A}$ asks a $\mathbf{KeyGen_C}$, $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query to $\mathbf{C}$, such queries are handled honestly, except that in order to handle each query, instead of computing $(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$, $\mathcal{B}$ uses the values sampled in the previous step (as it does not know *pwd*).

- All queries asked by $\mathcal{A}$ to $\mathbf{T}$ are handled honestly, except that after step 2 of $\mathcal{H}_{\mathbf{0}}$, whenever $\mathcal{A}$ submits a *handle* to $\mathbf{T}$ (either as part of its last message in **KeyGen** queries or as the first message in $\mathbf{PK_C}$ and $\mathbf{Sign_C}$ queries) and there exists a previously asked random oracle query for a password *pwd* such that $(\cdot, handle) = \mathbf{RO}_\kappa(pwd)$, then $\mathcal{B}$ submits *pwd* as a guess to its **Guess** oracle: if the response is 1, then $\mathcal{B}$ can halt (and **ExpGuess** outputs 1); otherwise the simulation continues honestly.

- If $\mathcal{A}$ halts (with or without outputting a forgery), then $\mathcal{B}$ halts as well (and **ExpGuess** outputs 0).

Notice that until $\mathcal{B}$ halts, the view of $\mathcal{A}$ has the same distribution as in $\mathcal{H}_{\mathbf{1}}$. The only change in $\mathcal{A}$'s view is that, instead of computing $(ek^*, handle^*) \leftarrow \mathbf{RO}_\kappa(pwd)$ (where *pwd* is the password chosen by the challenger), $\mathcal{B}$ samples such values uniformly at random and independently from *pwd*. The only way that $\mathcal{A}$ can notice such difference is if it either guesses $handle^*$ and asks a query to $\mathbf{T}$ where it uses such value (which can happen with at most negligible probability, and we implicitly condition over this event not happening) or if, after having output PWD (i.e. after step 2 of the original **ExpForgePwdGuess** experiment), $\mathcal{A}$ asks a random oracle query for *pwd* and sends the resulting output *handle* to $\mathbf{T}$ as part of a query. However, in this case, $\mathcal{B}$ will halt and win the experiment. Moreover, if the simulated $\mathcal{H}_{\mathbf{1}}$ execution outputs GUESSED, then $\mathcal{B}$ will cause **ExpGuess** to output 1. This proves that $\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] < \Pr[\mathbf{ExpGuess}_{\mathcal{B}}(1^\kappa) = 1] + \mu(\kappa)$ (the μ function accounts for the probability that $\mathcal{A}$ guesses $handle^*$). Moreover, the number of guesses that $\mathcal{B}$ makes to its oracle is upper bounded by the number of interactions with $\mathbf{T}$ made by $\mathcal{A}$ after step 2, from which by Lemma 5 we have that $\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] < \frac{q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa)$, which is what we wanted to prove.

2. For any adversary $\mathcal{A}$, if the output of an execution of $\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa)$ is 1, then the forgery (pk', m', σ') output by the adversary must be such that there is a record (pk, s) in g where $pk = pk'$ and $m \notin s$. We call the index i of any such record (i.e. its position in the list g) *important* for the execution.

Assume by contradiction that there exists an adversary $\mathcal{A}$ such that $\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]$ is non negligible. We can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ which contradicts the unforgeability for C of the underlying Threshold Signature scheme. Let $d(\kappa)$ be an upper bound on the number of **KeyGen** queries asked by $\mathcal{A}$. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $\mathcal{H}_{\mathbf{1}}$. First, it samples $i \leftarrow_R \{1, \ldots, d(\kappa)\}$, and its success probability will depend on such index i being important as defined above. Then it handles queries asked by $\mathcal{A}$ as follows.

- When $\mathcal{A}$ outputs a distribution $\texttt{PWD}$, $\mathcal{B}$ samples $pwd \leftarrow \texttt{PWD}$ and $(ek^*, handle^*) \leftarrow \mathbf{RO}_\kappa(pwd)$.

- All queries to the **T** and **C** oracles (of any kind) asked before the i-th $\mathbf{KeyGen_C}$ query to the **C** oracle are simulated as the honest challenger would in an execution of $\mathcal{H}_{\mathbf{1}}$.

- The i-th $\mathbf{KeyGen_C}$ query by the adversary to **C** is handled differently. First, $\mathcal{B}$ interacts with its own challenger in an execution of **TS**.**Gen** where the challenger runs **TS**.**GT** and $\mathcal{B}$ runs the **TS**.**GC** algorithm honestly. From this interaction, $\mathcal{B}$ obtains sk^*_C, pk^*; it returns pk^* to $\mathcal{A}$ and stores the secret key share sk^*_C (note that $\mathcal{B}$ does not learn the challenger's secret key share).

- After the i-th $\mathbf{KeyGen_C}$ query, all subsequent queries to the **T** oracle are simulated as in an execution of $\mathcal{H}_{\mathbf{1}}$. In particular, if the adversary interacts with the oracle **T** and sends it $handle^*$ (either as part of the last message in a $\mathbf{KeyGen_C}$ query or as the first message of a $\mathbf{PK_C}$ or $\mathbf{Sign_C}$ query), then $\mathcal{B}$ halts and outputs $\texttt{GUESSED}$.

- $\mathbf{Sign_C}$ and $\mathbf{PK_C}$ queries to the **C** oracle asked after the i-th $\mathbf{KeyGen_C}$ query but before the $(i+1)$-th $\mathbf{KeyGen_C}$ query are handled as follows. When $\mathcal{A}$ asks a $\mathbf{Sign_C}$ query on input m, then $\mathcal{B}$ interacts with its challenger in an execution of **TS**.**Sign** on input m (where $\mathcal{B}$ runs $\mathbf{TS}.\mathbf{SC}(sk^*_C, m)$). From this interaction, $\mathcal{B}$ obtains a signature σ, which it returns to $\mathcal{A}$. Analogously, $\mathbf{PK_C}$ queries from $\mathcal{A}$ are handled by simply returning pk^* to $\mathcal{A}$.

  All other queries to the **C** oracle (from the $(i+1)$-th $\mathbf{KeyGen_C}$ query onwards) are simulated as in $\mathcal{H}_{\mathbf{1}}$.

- When $\mathcal{A}$ halts and outputs a forgery (pk', m', σ'), then if $\mathbf{Ver}(pk', m', \sigma') = 1$, $pk' = pk^*$ and $m \notin s$ (where the i-th record in g is (pk^*, s)), then $\mathcal{B}$ outputs m', σ' as a forgery and halts; otherwise $\mathcal{B}$ halts without producing any forgery.

Notice that $\mathcal{A}$'s view in this experiment has the same distribution as its view in $\mathcal{H}_{\mathbf{1}}$. This is because the only difference between the two is that in some of the queries to the **C** oracle, $\mathcal{B}$ provides answers to $\mathcal{A}$ through its challenger instead that through an interaction with **T**. However, since $\mathcal{B}$'s challenger implements the signature scheme honestly, and $\mathcal{A}$ cannot query **T** on input $handle^*$, this difference does not affect $\mathcal{A}$'s view.

Moreover, we have that if the simulated $\mathcal{H}_{\mathbf{1}}$ execution outputs 1 and $\mathcal{B}$ guessed the index i correctly (which happens with probability at least $\frac{1}{d(\kappa)}$ conditioned on the former event), then $\mathcal{B}$'s output will also be a valid forgery for **TS**.**ForgeC**.

Thus we have that $\Pr[\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa) = 1] \leq d(\kappa) \cdot \Pr[\mathbf{TS}.\mathbf{ForgeC}^{TS}_{\mathcal{B}}(1^\kappa) = 1]$, which contradicts the unforgeability for the client of the Threshold Signature scheme. □

A.4 Proof of Theorem 21

Proof. We prove the theorem through an hybrid argument. Assume by contradiction that there exists a *compliant* adversary $\mathcal{A}$ that distinguishes between $\mathbf{ExpNonSignal}^{\mathsf{2FS},0}$ and $\mathbf{ExpNonSignal}^{\mathsf{2FS},1}$. Let $d(\kappa)$ be a polynomial upper bound on the number of queries asked by $\mathcal{A}$. We define the following hybrids:

- $\mathcal{H}_{\mathbf{0}}$: This is the same as $\mathbf{ExpNonSignal}^{2FS,0}_{\mathcal{A}}$.

- $\mathcal{H}_{\mathbf{1,i}}$: For each $i = 0, \dots, d(\kappa)$, $\mathcal{H}_{\mathbf{1,i}}$ is similar to $\mathcal{H}_{\mathbf{0}}$, with the following modifications. The first $d(\kappa) - i$ queries (either **KeyGen**, **PK** or **Sign**) from the adversary are answered by interacting with the circuit Π as in $\mathcal{H}_{\mathbf{0}}$. In addition, for each such **KeyGen** query by the adversary on input *pwd*, where $\mathcal{A}$ receives from the challenger as an answer *pk*, the challenger creates a record (pwd, pk).

  All other queries are handled as follows: **KeyGen** queries for *pwd* are handled by running $(\cdot, \cdot, pk) \leftarrow$ **TS**.**Gen**(1^κ) and returning *pk* to $\mathcal{A}$ (the challenger also creates a record (pwd, pk)). **PK** queries for *pwd* are answered by returning to $\mathcal{A}$ the *pk* from the last (pwd, pk) record created, or $\perp$ if no such record exists. **Sign** queries for *pwd* and *m* are answered by looking up the *pk* from the last (pwd, pk) record created and sampling a signature uniformly at random from the set $\{\sigma : \mathbf{Ver}(pk, m, \sigma) = 1\}$ (or returning $\perp$ if no such record exists). Note that such sampling requires exponential time. The random oracle is implemented honestly (by sampling values uniformly and storing them for consistency).

- $\mathcal{H}_{\mathbf{2,i}}$: For each $i = 0, \dots, d(\kappa)$, $\mathcal{H}_{\mathbf{2,i}}$ is built from $\mathcal{H}_{\mathbf{3}}$, with the analogous modifications of $\mathcal{H}_{\mathbf{1,d(\kappa)-i}}$. In particular, the first i queries (either **KeyGen**, **PK** or **Sign**) from the adversary are answered by interacting with an honest Token Oracle as in $\mathcal{H}_{\mathbf{3}}$ (the challenger keeps the same (pwd, pk) records described in the previous hybrids). The rest of the queries are simulated either by running the **TS**.**Gen** algorithm or by sampling signatures uniformly at random as in the previous hybrids.

- $\mathcal{H}_{\mathbf{3}}$: This is the same as $\mathbf{ExpNonSignal}_{\mathcal{A}}^{2FS,1}$.

Since the number of hybrids is polynomial, and $\mathcal{A}$ by assumption $\mathcal{A}$ distinguishes between $\mathcal{H}_{\mathbf{0}}$ and $\mathcal{H}_{\mathbf{3}}$ with non negligible probability, then there must exists two consecutive hybrids such that $\mathcal{A}$ can distinguish between them with non negligible probability. First, note that by construction the couples of hybrids $\mathcal{H}_{\mathbf{0}} = \mathcal{H}_{\mathbf{1,0}}$, $\mathcal{H}_{\mathbf{1,d(\kappa)}} = \mathcal{H}_{\mathbf{2,0}}$, $\mathcal{H}_{\mathbf{2,d(\kappa)}} = \mathcal{H}_{\mathbf{3}}$ are identical. Therefore $\mathcal{A}$ must be able to distinguish between $\mathcal{H}_{\mathbf{1,i}}$ and $\mathcal{H}_{\mathbf{1,i+1}}$ or $\mathcal{H}_{\mathbf{2,i}}$ and $\mathcal{H}_{\mathbf{2,i+1}}$ for some specific $i \in \{1, \dots, d(\kappa) - 1\}$. We will show that the first case leads to a contradiction; the second case is analogous (just substitute Π with an honest token oracle in the reasoning below, and note that such an honest token implementation never aborts and always returns signatures w.r.t. the expected public keys, and as such satisfies the restrictions of the Non-Signalling definition).

Note that the view of the adversary, the internal state of the challenger and of the circuit Π have exactly the same distribution in $\mathcal{H}_{\mathbf{1,i}}$, $\mathcal{H}_{\mathbf{1,i+1}}$ and $\mathbf{ExpNonSignal}_{\mathcal{A}}^{\mathrm{2FS},1}$ up to the point where the $(d(\kappa)-i)$-th query is asked by the adversary (but not answered). Consider such a prefix of an execution, we say that such a prefix is *compliant* if all the $d(\kappa) - i - 1$ interactions (originated from queries by $\mathcal{A}$) between the challenger and Π that are part of the prefix are compliant (as described in Definition 15) and, moreover, with overwhelming probability (over the randomness used by the client and any extra randomness which Π might use) given such prefix the $(d(\kappa) - i)$-th interaction will also be compliant

Let us now fix, for each κ, the compliant prefix of the execution p_κ which maximizes the distinguishing probability of the adversary conditioned on such prefix. Since $\mathcal{A}$ is compliant, with overwhelming probability an execution of $\mathcal{H}_{\mathbf{1,i+1}}$ will produce a compliant prefix, and so if $\mathcal{A}$ distinguishes $\mathcal{H}_{\mathbf{1,i}}$ from $\mathcal{H}_{\mathbf{1,i+1}}$ with non negligible probability, its success probability must be non negligible even conditioned on such maximizing prefix. More formally, there must exist a polynomial q such that for infinitely many κ,

$$|\Pr[\mathcal{H}_{\mathbf{1,i}}(\mathcal{A}, 1^\kappa) = 1 \mid p_\kappa] - \Pr[\mathcal{H}_{\mathbf{1,i+1}}(\mathcal{A}, 1^\kappa) = 1 \mid p_\kappa]| > \frac{1}{q(\kappa)} \tag{8}$$

Consider the last (i.e. the $(d(\kappa) - i)$-th) query in each prefix p_κ (note again that such query is asked in the prefix, but not answered). We will consider different three different cases, depending on the type of such query. Since the above equation holds for infinitely many κ, there must exist at least one case such that for infinitely many κ, equation 8 holds and the last query of p_κ belongs to such case. We will show that each case leads to a contradiction.

1. If for infinitely many κ the last query in p_κ is a **KeyGen** query (and equation 8 holds), we will build an adversary that contradicts the first of the Non-Signalling properties for the Threshold Signature scheme.

2. If for infinitely many κ the last query in p_κ is a **PK** queries, or a **Sign** query on input a password *pwd* such that the challenger does not have a (pwd, pk) record, we will contradict the assumption that $\mathcal{A}$ is compliant.

3. If for infinitely many κ the last query in p_κ is a **Sign** query on input a password *pwd* such that the challenger does have a (pwd, pk) record, we will build an adversary that contradicts the second of the Non-Signalling properties for the Threshold Signature scheme.

1. We will first assume that, for infinitely many κ, the $(d(\kappa) - i)$-th query the adversary asks in the prefix considered is a **KeyGen** query.

To answer such query, in $\mathcal{H}_{\mathbf{1},\mathbf{i}}$ the challenger will run $\mathbf{TS}.\mathbf{GT}$ interacting with Π to obtain *pk*, while in $\mathcal{H}_{\mathbf{1},\mathbf{i+1}}$ it will compute *pk* by itself through an honest execution of $\mathbf{TS}.\mathbf{Gen}(1^\kappa)$. In both cases it would return *pk* to $\mathcal{A}$. Given this adversary and the maximizing prefix of the execution fixed above, we can build an adversary $\mathcal{B}$ that contradicts the first Non-Signalling property of Threshold Signature scheme. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ outputs Π (in the state it had in the fixed execution prefix p_κ), and receives in response a public key *pk*.
- It continues running $\mathcal{A}$ from the prefix above by answering its $(d(\kappa) - i)$-th query with *pk*.
- All subsequent queries are answered as in hybrid $\mathcal{H}_{\mathbf{1},\mathbf{i}}$ (or, equivalently, $\mathcal{H}_{\mathbf{1},\mathbf{i+1}}$).
- When $\mathcal{A}$ halts and outputs a bit b, $\mathcal{B}$ outputs the same.

First of all, note that since the prefix of the execution we are considering is compliant, then $\mathcal{B}$'s program satisfies the condition that $\Pr[\langle \cdot, pk; \cdot\rangle \leftarrow \langle \mathbf{TS}.\mathbf{GC}(1^\kappa); \Pi\rangle : pk \neq \perp] > 1 - \mu(\kappa)$ (i.e., Π's next interaction with the challenger will not be compliant with at most negligible probability) and so it will be a valid adversary for $\mathbf{TS}.\mathbf{NS1}$. Moreover, when $\mathcal{B}$ is running in $\mathbf{TS}.\mathbf{NS1}^{\mathsf{TS},0}$, then $\mathcal{A}$'s view has the same distribution as in an execution of $\mathcal{H}_{\mathbf{1},\mathbf{i}}$ (conditioned on p_κ), while if the public key $\mathcal{B}$ receives is sampled uniformly (as in $\mathbf{TS}.\mathbf{NS1}^{\mathsf{TS},1}$), then $\mathcal{A}$'s view is consistent with $\mathcal{H}_{\mathbf{1},\mathbf{i+1}}$. So, if $\mathcal{A}$ can distinguish between the two hybrids (conditioned on p_κ) then $\mathcal{B}$ can also distinguish between the two distributions with the same advantage: for infinitely many κ,

$$\frac{1}{q(\kappa)} < |\Pr[\mathcal{H}_{\mathbf{1},\mathbf{i}}(\mathcal{A}) = 1 \mid p_\kappa] - \Pr[\mathcal{H}_{\mathbf{1},\mathbf{i+1}}(\mathcal{A}) = 1 \mid p_\kappa]|$$
$$< |\Pr[\mathbf{TS}.\mathbf{NS1}_{\mathcal{B}}^{\mathsf{2FS},0}(1^\kappa) = 1] - \Pr[\mathbf{TS}.\mathbf{NS1}_{\mathcal{B}}^{\mathsf{2FS},1}(1^\kappa) = 1]|$$

which contradicts the first of the first Non-Signalling property of the Threshold Signature scheme. Note that $\mathcal{B}$'s running time is exponential, as it has to sample signatures uniformly at random to answer some of $\mathcal{A}$'s queries, but this does not affect the validity of our reduction as the Non-Signalling definition for Threshold Signature schemes holds even against exponential time adversaries.

2. Consider now the case where, for infinitely many κ, the $(d(\kappa) - i)$-th query the adversary asks in the fixed prefix p_κ is a **PK** query, or a **Sign** query on input a password *pwd* such that the challenger does not have a (pwd, pk) record.

For queries (**PK** and **Sign**) where the input is a password such that no record exists, $\mathcal{A}$ will receive $\perp$ as an answer in $\mathcal{H}_{\mathbf{1},\mathbf{i+1}}$, so the only way that its view can be different in $\mathcal{H}_{\mathbf{1},\mathbf{i}}$ is if Π causes the result of the query to be different from $\perp$. Similarly, for **PK** queries on input *pwd* where a record (pwd, pk) exists, in $\mathcal{H}_{\mathbf{1},\mathbf{i+1}}$ the adversary will receive as an answer *pk* (more specifically, the one from the latest such record created), so the only way that $\mathcal{A}$'s view can be different in $\mathcal{H}_{\mathbf{1},\mathbf{i}}$ is if Π causes the result of the query to be a *pk* different from *pk* (or $\perp$). However, in this were to happen with more than negligible probability, it would imply that p_κ is not compliant, which is a contradiction.

3. Finally, consider the case where the $(d(\kappa) - i)$-th query the adversary asks in the fixed prefix is a signature query for *pwd*, and let (pwd, pk) be the most recent record for *pwd* that the challenger created.

To answer such query, in $\mathcal{H}_{\mathbf{1,i+1}}$ the challenger will sample a signature at random from the set $\{\sigma : \mathbf{Ver}(pk, m, \sigma) = 1\}$. Instead, in $\mathcal{H}_{\mathbf{1,i}}$ the challenger will compute $ek, handle \leftarrow \mathbf{RO}_\kappa(pwd)$, send *handle* to Π (as part of a **Sign** query), decrypt the ciphertext received in return to obtain a key share sk_C, and (assuming Π does send a ciphertext and decryption succeeds) run $\mathbf{TS}.\mathbf{SC}(sk_C, m)$ interacting with Π to obtain σ, which is returned to $\mathcal{A}$ as an answer to the query.

With a reasoning analogous to case 2, we can conclude that Π will abort before sending a ciphertext or decryption will fail with at most negligible probability (otherwise p_κ would not be compliant). Therefore, with a reasoning analogous to the one at the beginning of the proof, we can consider all the possible extensions p'_κ of the prefix p_κ up to the point where Π returns a ciphertext c to the challenger, and pick the compliant one which again maximizes the success probability of the adversary. It has to be that:

$$\frac{1}{q(\kappa)} < |\Pr[\mathcal{H}_{\mathbf{1,i}}(\mathcal{A}) = 1 \mid p_\kappa] - \Pr[\mathcal{H}_{\mathbf{1,i+1}}(\mathcal{A}) = 1 \mid p_\kappa]| < \\ |\Pr[\mathcal{H}_{\mathbf{1,i}}(\mathcal{A}, 1^\kappa) = 1 \mid p'_k] - \Pr[\mathcal{H}_{\mathbf{1,i+1}}(\mathcal{A}, 1^\kappa) = 1 \mid p'_k]|$$

Since this new extended prefix is compliant, it must be that in such prefix the challenger successfully decrypts c and recovers a secret key sk_C.

Given $\mathcal{A}$ and p'_κ, analogously as in case 1, we can build an adversary $\mathcal{B}$ that contradicts the second Non-Signalling property of Threshold Signature scheme. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ outputs sk_C,m and Π (in the state it had in p'_κ), and receives in response a signature σ.
- It continues running $\mathcal{A}$ from the prefix above by answering its $(d(\kappa) - i)$-th query with σ.
- All subsequent queries are answered as in hybrid $\mathcal{H}_{\mathbf{1,i}}$ (or, equivalently, $\mathcal{H}_{\mathbf{1,i+1}}$).
- When $\mathcal{A}$ halts and outputs a bit b, $\mathcal{B}$ outputs the same.

Note that since the prefix p'_κ is compliant, then $\mathcal{B}$'s program satisfies $\Pr[\langle\sigma; \cdot\rangle \leftarrow \langle\mathbf{TS}.\mathbf{SC}(1^\kappa); \Pi\rangle : \mathbf{Ver}(pk, m, \sigma) = 1] > 1 - \mu(\kappa)$ (i.e., it satisfies the restrictions of $\mathbf{TS}.\mathbf{NS2}$). Moreover, when $\mathcal{B}$ is running in $\mathbf{TS}.\mathbf{NS2}^{\mathsf{2FS},0}$, then $\mathcal{A}$'s view has the same distribution as in an execution of $\mathcal{H}_{\mathbf{1,i}}$ (conditioned on p'_κ), while if $\mathcal{B}$ receives a signature sampled at random (as in $\mathbf{TS}.\mathbf{NS2}^{\mathsf{2FS},1}$), then $\mathcal{A}$'s view is consistent with $\mathcal{H}_{\mathbf{2,i+1}}$. So, if $\mathcal{A}$ can distinguish between the two hybrids then $\mathcal{B}$ can also distinguish between the two distributions with the same advantage, which is a contradiction. Again, $\mathcal{B}$'s running time might be exponential, but the definition of Non-Signalling holds even against exponential time adversaries. □

A.5 Proof of Theorem 22

Proof. We prove the theorem through an hybrid argument. We define the following hybrids:

- $\mathcal{H}_{\mathbf{0}}$: Defined as $\mathbf{ExpForgeTokMan}_{\mathcal{A}}^{\mathsf{2FS}}(1^\kappa)$
- $\mathcal{H}_{\mathbf{1}}$: Defined from $\mathcal{H}_{\mathbf{0}}$ but where to answer $\mathcal{A}$'s queries, instead of interacting with Π, the challenger interacts with an honestly implemented token oracle T.

To prove the theorem, it is sufficient to show that for any PPT $\mathcal{A}$ there exists a negligible function μ such that:

$$|\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{9}$$

$$|\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{10}$$

We will prove that each condition holds separately.

1. Since Threshold Signature scheme is Non-Signalling, by Theorem 21 we have that the Two Factor Signature scheme construction is also Non-Signalling. Assume by contradiction that there exists an adversary $\mathcal{A}$ such that $\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]|$ is non negligible. We can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ which contradicts the Non-Signalling property of the Two Factor Signature scheme. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$ by simulating for it an execution of $\mathcal{H}_{\mathbf{0}}$. When $\mathcal{A}$ outputs a circuit Π, $\mathcal{B}$ forwards such program to its own challenger. Then, all queries by $\mathcal{A}$ are handled by forwarding such queries to its own challenger and returning to $\mathcal{A}$ the challenger's response.

- When $\mathcal{A}$ halts, $\mathcal{B}$ computes and outputs the output that $\mathcal{H}_{\mathbf{0}}$ would have in the simulated execution. In other words, $\mathcal{B}$ outputs 1 if the adversary outputs a forgery which would make $\mathcal{H}_{\mathbf{0}}$ output 1, and 0 otherwise.

Note that if $\mathcal{A}$ is compliant for **ExpForgeTokMan**, then $\mathcal{B}$ is also compliant for **ExpNonSignal** as both $\mathcal{A}$ and $\mathcal{B}$ output programs with the same distribution and make the same queries. Moreover, if $\mathcal{B}$ is running in an execution $\mathbf{ExpNonSignal}^{\mathtt{2FS},0}$ (resp. $\mathbf{ExpNonSignal}^{\mathtt{2FS},1}$), then $\mathcal{A}$'s view has the same distribution as in $\mathcal{H}_{\mathbf{0}}$ (resp. $\mathcal{H}_{\mathbf{1}}$).

Therefore $\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]| < |\Pr[\mathbf{ExpNonSignal}_{\mathcal{B}}^{\mathtt{2FS},0}(1^\kappa) = 1] - \Pr[\mathbf{ExpNonSignal}_{\mathcal{A}}^{\mathtt{2FS},1}(1^\kappa) = 1]|$ which contradicts the Non-Signalling property of `2FS`.

2. This reduction is similar to the proof of Theorem 18.

During an execution of the $\mathcal{H}_{\mathbf{1}}$, we say that the forgery (pk', m', σ') (produced by the adversary at the end of the experiment) is *associated with* the i-th **KeyGen** query if pk' is equal to the public key returned to the adversary during such **KeyGen** query.

Assume by contradiction that there exists an adversary $\mathcal{A}$ such that $\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]$ is non negligible. We can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ which contradicts the unforgeability for C of the underlying Threshold Signature scheme. Let $d(\kappa)$ be an upper bound on the number of **KeyGen** queries asked by $\mathcal{A}$. $\mathcal{B}$ works as follows:
$\mathcal{B}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $\mathcal{H}_{\mathbf{1}}$. First, it samples $i \leftarrow_R \{1, \ldots, d(\kappa)\}$, and its success probability will depend on the forgery output by $\mathcal{A}$ being associated with the i-th **KeyGen** query in the simulated execution. Then it handles queries by $\mathcal{A}$ as follows.

- All queries (of any kind) before the i-th **KeyGen** query are simulated as in $\mathcal{H}_{\mathbf{1}}$, i.e. by using $\perp$ as the initial state for a token oracle $\mathbf{T}$ and responding to $\mathcal{A}$'s queries by executing the honest algorithms interacting with $\mathbf{T}$ (and updating $\mathbf{T}$'s state as a result of **KeyGen** queries).

- The i-th **KeyGen** query by the adversary is handled differently. Let pwd^* be the password which $\mathcal{A}$ supplies as input for this query. First, $\mathcal{B}$ interacts with its own challenger in an execution of **TS**.**Gen**, where the challenger runs **TS**.**GT** and $\mathcal{B}$ plays the role of C, and obtains sk_C^*, pk^* as output. $\mathcal{B}$ stores pwd^*, sk_C^*, pk^* and returns pk^* to $\mathcal{A}$.

- After the i-th **KeyGen** query, when $\mathcal{A}$ asks a **Sign** query on input the same password pwd^* from the previous step and any message m, then $\mathcal{B}$ asks its challenger for a **Sign** query for m, where $\mathcal{B}$ executes $\mathbf{TS}.\mathbf{SC}(sk_C^*, m)$ and the challenger plays the role of T. As a result, $\mathcal{B}$ obtains a signature which is returned to $\mathcal{A}$. Analogously, **PK** queries on input pwd^* are handled by returning to $\mathcal{A}$ the recorded pk^*. All other **Sign** and **PK** queries (where $pwd \neq pwd^*$) are simulated honestly. **KeyGen** queries are also simulated honestly. In addition, if after the i-th **KeyGen** query $\mathcal{A}$ asks another **KeyGen** query using the recorded pwd^* as input, then $\mathcal{B}$ continues by simulating all subsequent queries honestly (using the token oracle $\mathbf{T}$ instead of its challenger), including the ones for pwd^*.

- When $\mathcal{A}$ outputs a forgery (pk', m', σ'), then if $pk' = pk^*$, $\mathcal{B}$ outputs (m', σ') as a forgery to its challenger; otherwise $\mathcal{B}$ halts without producing any forgery.

Notice that $\mathcal{A}$'s view in this experiment has the same distribution as its view in $\mathcal{H}_{\mathbf{1}}$. This is because the only difference between the two is that in some of the queries by the adversary where $pwd = pwd^*$, $\mathcal{B}$ provides answers to $\mathcal{A}$ by interacting with its challenger instead of with the $\mathbf{T}$ oracle; since both implement the algorithms honestly and using uniform randomness, this difference does not affect $\mathcal{A}$'s view.

Moreover, we have that if the simulated $\mathcal{H}_{\mathbf{1}}$ execution outputs 1 and $\mathcal{B}$ guessed the index i correctly (which happens with probability $\frac{1}{d(\kappa)}$ conditioned on the former event), then $\mathcal{B}$'s output will also be a valid forgery for $\mathbf{TS}.\mathbf{ForgeC}$ (since $(pk', m') \notin s$ for $\mathcal{H}_{\mathbf{1}}$, then $\mathcal{B}$ must have never queried his challenger on input m').

Thus we have that $\Pr[\mathbf{ExpForgeC}_{\mathcal{A}}^{2FS}(1^\kappa) = 1] \leq d(\kappa) \cdot \Pr[\mathbf{TS}.\mathbf{ForgeC}_{\mathcal{B}}^{TS}(1^\kappa) = 1]$, which contradicts the unforgeability against the client of the Threshold Signature scheme.

□

A.6 Proof of Theorem 27

The proof is structured as an hybrid argument. Starting from an execution of $\mathbf{ExpUnlink}^{\Pi, b_P, b_T}$, we can introduce a first hybrid where the experiment is aborted if the adversary sends to any of the tokens a handle derived as a hash of one of the two passwords sampled by the challenger. The probability of $\mathcal{A}$ noticing this difference can be bounded by $\frac{2q(\kappa)}{2^{m(\kappa)}}$ (which bounds the probability that $\mathcal{A}$ can guess one of the passwords) through a reduction to Lemma 6. Then, we notice that in this hybrid the view of the adversary does not depend on any of the two bits b_P, b_T, and so we can switch to an hybrid where we use b'_P, b'_T instead, and finally to an hybrid where we remove the abort condition (the adversary also has at most an at most $\frac{2q(\kappa)}{2^{m(\kappa)}}$ chance of distinguishing here). The last hybrid is a standard execution of $\mathbf{ExpUnlink}^{\Pi, b'_P, b'_T}$, so the advantage of the adversary can be bounded by $2 \cdot \frac{2q(\kappa)}{2^{m(\kappa)}}$ (plus a negligible amount due to possible collisions in the random oracle).

Proof. We prove the theorem through an hybrid argument. Let b_P, b_T, b'_P, b'_T be any 4 bits, with both $(b_P, b_T) \neq (0, 0)$ and $(b'_P, b'_T) \neq (0, 0)$. We define the following hybrids:

- $\mathcal{H}_{\mathbf{0}}$: Defined as $\mathbf{ExpUnlink}_{\mathcal{A}}^{\Pi, b_P, b_T}(1^\kappa)$
- $\mathcal{H}_{\mathbf{1}}$: Defined from $\mathcal{H}_{\mathbf{0}}$, where in addition the experiment is aborted (with a special output `GUESSED`) if the adversary, during any interaction with any of the two tokens, sends them a *handle* which is equal to the second half of the answer $\mathcal{A}$ received from querying the random oracle on either pwd_0 or pwd_1 (i.e. $(\cdot, handle) = \mathbf{RO}_\kappa(pwd_0)$ or $(\cdot, handle) = \mathbf{RO}_\kappa(pwd_1)$), where pwd_0, pwd_1 are the two passwords sampled by the challenger in step 2.
- $\mathcal{H}_{\mathbf{2}}$: Defined from $\mathcal{H}_{\mathbf{3}}$, where we add the same abort condition of $\mathcal{H}_{\mathbf{1}}$.
- $\mathcal{H}_{\mathbf{3}}$: Defined as $\mathbf{ExpUnlink}_{\mathcal{A}}^{\Pi, b'_P, b'_T}(1^\kappa)$

To prove the theorem, it is sufficient to show that for any PPT $\mathcal{A}$ there exists a negligible function μ such that:

$$|\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]| < \frac{2q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa) \tag{11}$$

$$|\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^\kappa) = 1]| < \mu(\kappa) \tag{12}$$

$$|\Pr[\mathcal{H}_{\mathbf{2}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{3}}(\mathcal{A}, 1^\kappa) = 1]| < \frac{2q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa) \tag{13}$$

We will prove the first and second conditions separately; the proof of the third one is analogous to the first one (it is enough to substitute b'_P, b'_T for b_P, b_T) and thus is omitted.

1. Analogously to the first case of the proof of theorem 19, one can show that

$$\left|\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = 1] - \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = 1]\right| \leq \Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] \tag{14}$$

Therefore to prove that equation 1 holds it is enough to show that $\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] < \frac{2q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa)$. We will prove that this is the case by leveraging Lemma 6. For any PPT adversary $\mathcal{A}$ for $\mathcal{H}_{\mathbf{1}}$, let's define an adversary $\mathcal{B}$ for **ExpGuessTwo** as follows:
$\mathcal{B}^{\mathbf{Guess}(\cdot)}(1^\kappa)$:

- $\mathcal{B}$ runs $\mathcal{A}$ by simulating for it an execution of $\mathcal{H}_{\mathbf{1}}$. All initial queries from step 1 are answered as the honest challenger would. $\mathcal{B}$ also simulates for $\mathcal{A}$ the random oracle $\mathbf{RO}_\kappa$ by sampling answers to $\mathcal{A}$'s queries uniformly at random and storing them in a table so that the same query always receives the same answer.

- When $\mathcal{A}$ outputs a distribution `PWDS`, $\mathcal{B}$ forwards such distribution to its challenger. It then samples $(sk_C^\alpha, sk_T^\alpha, pk^\alpha) \leftarrow \mathbf{TS}.\mathbf{Gen}(1^\kappa)$ and $(sk_C^\beta, sk_T^\beta, pk^\beta) \leftarrow \mathbf{TS}.\mathbf{Gen}(1^\kappa)$ and returns pk_α, pk_β to $\mathcal{A}$. From this point on, signing queries for O^α and O^β are computed by using the corresponding secret keys. All subsequent queries to $\mathbf{T}^0$, $\mathbf{T}^1$ are still simulated as the honest challenger would, but with the following modification: whenever $\mathcal{A}$ sends to one of the token oracles a *handle* (as part of any **KeyGen**, **PK** or **Sign** query), the challenger checks if the adversary has made a random oracle query for some *pwd* such that $\mathbf{RO}_\kappa(pwd) = (\cdot, handle)$. If so, it submits such *pwd* as a query to its **Guess** oracle: if the response is 1, then $\mathcal{B}$ can halt (and **ExpGuessTwo** outputs 1); otherwise the simulation continues.

- If $\mathcal{A}$ halts (with or without outputting a bit), then $\mathcal{B}$ halts as well (and **ExpGuessTwo** outputs 0).

Notice that until $\mathcal{B}$ halts, the distribution of the view of $\mathcal{A}$ is statistically close to the distribution of its view in $\mathcal{H}_{\mathbf{1}}$. Indeed, the only difference between the two distributions is that in this experiment, instead of computing pk_α by interacting with $\mathbf{T}^0$ and pk_β by interacting with $\mathbf{T}^{b_T}$ as in $\mathcal{H}_{\mathbf{1}}$, $\mathcal{B}$ samples them at random on its own. Moreover, signatures returned by O^α and O^β are also computed by the challenger without using the oracles $\mathbf{T}^0, \mathbf{T}^{b_T}$. However, since in any execution of $\mathcal{H}_{\mathbf{1}}$, since $(b_P, b_T) \neq (0,0)$ and `PWDS` is such that $pwd_0 \neq pwd_1$, we have that $(pwd_0, b_0) \neq (pwd_{b_P}, b_T)$. This means that, unless $\mathbf{RO}_\kappa(pwd_0)$ and $\mathbf{RO}_\kappa(pwd_1)$ output the same *handle*, which happens with negligible probability, then pk_α and pk_β are also independently generated and similarly O^α and O^β will return valid signatures[7] w.r.t. pk_α, pk_β in any execution of $\mathcal{H}_{\mathbf{1}}$. Since public keys and oracle answers in the two cases have distributions which are statistically close (identical except for the probability of a random oracle collision), then the only way that $\mathcal{A}$ could notice that the state of such $\mathbf{T}$ oracles is inconsistent with pk_α, pk_β is by interacting with one such $\mathbf{T}$ oracle on input pwd_0 or pwd_1 and sending them a handle computed as $(\cdot, handle) \leftarrow \mathbf{RO}_\kappa(pwd_0)$ or $(\cdot, handle) \leftarrow \mathbf{RO}_\kappa(pwd_1)$. In this case, though, $\mathcal{B}$ would halt and **ExpGuessTwo** would output 1. This proves that $\Pr[\mathcal{H}_{\mathbf{1}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] < \Pr[\mathbf{ExpGuessTwo}_{\mathcal{B}}(1^\kappa) = 1] + \mu(\kappa)$ (where the negligible factor accounts for possible random oracle collisions). Moreover, the number of guesses that $\mathcal{B}$ makes to its oracle is upper bounded by the number of interactions between $\mathcal{A}$ and the two $\mathbf{T}$ oracles after step 2 ($\mathcal{A}$ can only send one *handle* per interaction with such oracles), from which by Lemma 6 we have that $\Pr[\mathcal{H}_{\mathbf{0}}(\mathcal{A}, 1^\kappa) = \texttt{GUESSED}] < \frac{2q(\kappa)}{2^{m(\kappa)}} + \mu(\kappa)$, which is what we wanted to prove.

2. A similar reasoning as in the end of the previous case shows that, if we condition on the absence of random oracle collisions, then the view of the adversary in the two hybrids is actually identical. pk_α and pk_β are sampled independently in both experiments, and if there are no collisions in the random oracle then the answers of O^α and O^β will be consistent with such public keys respectively. So the only way in which $\mathcal{A}$'s view might differ is if he was able to query one of the $\mathbf{T}$ oracles on one of the two passwords picked by the challenger, but in this case both experiments would be aborted. This proves that $\mathcal{H}_{\mathbf{1}}$ and $\mathcal{H}_{\mathbf{2}}$ are statistically close. □

[7] Otherwise, if for example $(b_P, b_T) = (0,0)$, in $\mathcal{H}_{\mathbf{0}}$ the challenger would be interacting twice with $\mathbf{T}^0$ on input the same password (which would mean that the information related to pk^α would be overwritten and never used to produce any signatures, so all subsequent Signing queries produced by both O^α and O^β would verify with respect to pk^β. A similar situation could happen even in the case where $(b_P, b_T) = (1,0)$ if $\mathbf{RO}_\kappa(pwd_0) = \mathbf{RO}_\kappa(pwd_1)$, but since we enforce that $pwd_0 \neq pwd_1$ the above collision can happen with at most negligible probability.