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

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

* 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.

has authorized it by pressing the physical button[3]. 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.

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 [16]. 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

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

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, 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 [14], 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 [13], 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).[4]

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.

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

[4] 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.

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 Related Work

Threshold Signatures Threshold signatures [4,7,15,2,1] 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.* [14] 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.

1.3 Organization of the paper

After introducing some notation in Section 2, we recall the definition of Threshold Signature scheme and introduce the Non-Signalling property in Section 3. Section 4 defines Two Factor Signature schemes and Section 5 presents our main construction and a sketch for some of the security proofs.

Due to lack of space, some of the security definitions (introduced earlier in the introduction), the full proofs of security, as well as the two modified TS schemes (based on Schnorr and EC-DSA) are deferred to the full version of this paper. There, we also discuss an additional useful *Unlinkability* property satisfied by our construction.

2 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.

3 Threshold Signature scheme

This section recalls the definition of a Threshold Signature scheme. 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 1. *A (2-out-of-2) Threshold Signature scheme consists of a tuple of distributed PPT algorithms defined as follows:*

- $\langle \mathbf{TS}.\mathbf{GC}(1^\kappa); \mathbf{TS}.\mathbf{GT}(1^\kappa)\rangle \to \langle sk_C, pk; sk_T, pk\rangle$ *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* $\mathbf{TS}.\mathbf{Gen}(1^\kappa) \to (sk_C, sk_T, pk)$ *as a compact expression for the above computation.*
- $\langle \mathbf{TS}.\mathbf{SC}(sk_C, m); \mathbf{TS}.\mathbf{ST}(sk_T, m)\rangle \to \langle \sigma; \perp\rangle$ *are two randomized algorithms interacting to produce as output a signature*[5] σ*. We use* $\mathbf{TS}.\mathbf{Sign}(sk_C, m, sk_T) \to \sigma$ *as as a compact expression for the above computation.*
- $\mathbf{TS}.\mathbf{Ver}(pk, m, \sigma) \to 0 \vee 1$ *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

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

The definitions of Unforgeability for the two parties (T and C) we require are quite standard and are deferred to the full version. In the following, we introduce a new security definition, which we call *Non-Signalling*. It 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 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

[5] 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.

$\mathcal{A}$ 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 2. *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 $\mathbf{TS.GC}$ 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 $\mathbf{TS.SC}$ 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 3. *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 ini-*

tial 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*

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

Security notions. We define five notions of security for a 2FS, all introduced in the introduction. Unforgeability for the Token, and Unforgeability with access to the Token are formalized in the full version. Here, we define Unforgeability for the Client, Unforgeability for the Token Manufacturer and Non-Signalling.

Definition 4 (Unforgeability for the Client). *Given a Two Factor Signature scheme 2FS* $= (\mathbf{KeyGen_C}, \mathbf{KeyGen_T}, \mathbf{PK_C}, \mathbf{PK_T}, \mathbf{Sign_C}, \mathbf{Sign_T}, \mathbf{Ver})$, *consider the following experiment between an adversary* $\mathcal{A}$ *and a challenger:*
$\mathbf{ExpForgeC}_{\mathcal{A}}^{2FS}(1^\kappa)$:

1. *The challenger runs the adversary* $\mathcal{A}$*, giving it access to a token oracle* $\mathbf{T}$ *(*$\mathcal{A}$ *is 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

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

The purpose of the two lists g and p in the experiment above is to ensure that either the adversary can cause the honest token to output a public key pk during a **PK** query which it did not output during a **KeyGen** query, or that all the forgeries returned by the adversary are w.r.t. public keys which were output by the honest token oracle, that the number of forgeries on m is greater than the number of signing queries which the challenger answered for m.

The next definition, Unforgeability for the Token Manufacturer, is formalized as an experiment where the adversary first outputs a stateful program Π, and then 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 1. 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 5. *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 PPT adversary $\mathcal{A}$ and a challenger, parameterized by a bit b:*
$\mathbf{ExpForgeTokMan}_{\mathcal{A}}^{2FS}(1^{\kappa})$:

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 $\mathbf{ExpForgeTokMan}^{2FS}$*, 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* $\mathbf{ExpForgeTokMan}^{\mathit{2FS}}_{\mathcal{A}}(1^\kappa)$ *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*

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

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 6. *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, parameterized by a bit b:*
$\mathbf{ExpNonSignal}^{\mathit{2FS},b}_{\mathcal{A}}(1^\kappa)$:

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 5.*

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):

- $\mathbf{KeyGen_C}(pwd) \rightarrow pk$: Run $\mathbf{TS}.\mathbf{GC}(1^\kappa)$ interacting with $\mathbf{KeyGen_T}$ and obtain (sk_C, pk) as the local output. Then, compute $(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$, $c \leftarrow \mathbf{SE}.\mathbf{E}(ek, (sk_C, pk))$ and send $(handle, c)$ to T. Output pk.
- $\mathbf{KeyGen_T}(s_T) \rightarrow s'_T$: Run $\mathbf{TS}.\mathbf{GT}(1^\kappa)$ interacting with $\mathbf{KeyGen_C}$ and obtain sk_T, pk as the local output. Then, receive $(handle, c)$ from $\mathbf{KeyGen_C}$, set $s'_T \leftarrow s_T.\mathbf{Add}(handle, (c, sk_T, pk))$ and output (s'_T, pk).
- $\mathbf{PK_C}(pwd)$: Compute $(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$, send $handle$ to $\mathbf{PK_T}$. Upon receiving c in response, compute $(sk_C, pk) \leftarrow \mathbf{SE}.\mathbf{D}(ek, c)$ and output pk.
- $\mathbf{PK_T}(s_T)$: Upon receiving $handle$ from $\mathbf{PK_C}$, retrieve from the state $(c, sk_C, pk) \leftarrow s_T.\mathbf{Find}(handle)$, send c to $\mathbf{PK_C}$ and output pk.
- $\mathbf{Sign_C}(pwd, m)$: Compute $(ek, handle) \leftarrow \mathbf{RO}_\kappa(pwd)$ and send $handle$ to $\mathbf{Sign_T}$. Upon receiving c in response, compute $(sk_C, pk) \leftarrow \mathbf{SE}.\mathbf{D}(ek, c)$, then execute $\mathbf{TS}.\mathbf{SC}(sk_C, m)$ (interacting with $\mathbf{Sign_T}$) and output the resulting σ.
- $\mathbf{Sign_T}(s_T, m)$: Upon receiving $handle$ from $\mathbf{PK_C}$, compute $(c, sk_C, pk) \leftarrow s_T.\mathbf{Find}(handle)$, send c to $\mathbf{Sign_C}$ and run $\mathbf{TS}.\mathbf{ST}(sk_T, m)$.
- $\mathbf{Ver}(pk, m, \sigma)$: Output $\mathbf{TS}.\mathbf{Ver}(pk, m, \sigma)$.

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

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

The security of the scheme is established by the following theorems. We provide a proof sketch for some of them, and defer the details to the full version.

Theorem 1. *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 output 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" (see the full version for details). $\mathcal{B}$ 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**. □

Theorem 2. *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.*

Theorem 3. *If the underlying Threshold Signature scheme is Unforgeable for the Client, the Two Factor Signature scheme described above is Unforgeable with Token Approval.*

Theorem 4. *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 5. *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. $\square$

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. Microchip. Atecc608a datasheet, 2018.
14. Antonio Nicolosi, Maxwell N Krohn, Yevgeniy Dodis, and David Mazieres. Proactive two-party signatures for user authentication. In *NDSS*, 2003.
15. Tal Rabin. A simplified approach to threshold and proactive RSA. In *Advances in Cryptology – CRYPTO 1998*, pages 89–104. Springer, 1998.
16. 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`].