# Sequential Digital Signatures for Cryptographic Software-Update Authentication

Bertram Poettering[1] and Simon Rastikian[1,2]

[1] IBM Research Europe – Zurich, Rüschlikon, Switzerland
[2] ETH Zurich, Zurich, Switzerland

**Abstract.** Consider a computer user who needs to update a piece of software installed on their computing device. To do so securely, a commonly accepted ad-hoc method stipulates that the old software version first retrieves the update information from the vendor's public repository, then checks that a cryptographic signature embedded into it verifies with the vendor's public key, and finally replaces itself with the new version. This updating method seems to be robust and lightweight, and to reliably ensure that no malicious third party (e.g., a distribution mirror) can inject harmful code into the update process. Unfortunately, recent prominent news reports (SolarWinds, Stuxnet, TikTok, Zoom, ...) suggest that nation state adversaries are broadening their efforts related to attacking software supply chains. This calls for a critical re-evaluation of the described signature based updating method with respect to the real-world security it provides against particularly powerful adversaries.

We approach the setting by formalizing a cryptographic primitive that addresses specifically the secure software updating problem. We define strong, rigorous security models that capture forward security (stealing a vendor's key today doesn't allow modifying yesterday's software version) as well as a form of self-enforcement that helps protecting vendors against coercion attacks in which they are forced, e.g. by nation state actors, to misuse or disclose their keys. We note that the common signature based software authentication method described above meets neither the one nor the other goal, and thus represents a suboptimal solution. Hence, after formalizing the syntax and security of the new primitive, we propose novel, efficient, and provably secure constructions.

## 1 Introduction

In August 2020, the US president signed an executive order requiring TikTok, a social media app of Chinese origin, to either be made unavailable on the US market or to be transferred to a new non-Chinese owner. The corresponding press statement reports that "credible evidence" indicates that the original producer of the app "might take action that threatens to impair the national security of the United States".[3] Four months later, the US Department of Justice publicly accused an executive of the company behind the telecommunication service Zoom to have, on behalf of the Chinese government, misused the Zoom app to "willingly commit crimes [... and ...] unlawful conspiracy [...] against US-based individuals", and concluded that "no company with significant business interests in China is immune from the coercive power of the Chinese Communist Party".[4] Opinions on the appropriateness of the press statements, and the actions taken, may be split, but the steps make a recent tendency of nation states evident, namely to question the harmlessness and innocence of software originating from other countries. The main perceived threat seems to be that another country's government might coerce its legitimate software vendors to embed backdoors or hidden espionage tools into their products that could be used against the own country. Without going into details, we mention two more attacks (with different configurations of attacking and attacked countries) where users received manipulated software over seemingly regular distribution channels: The well-known Stuxnet attack was conducted by manipulating the driver distribution scheme of the Windows operating system,[5] and the recently uncovered SolarWinds attacks centrally and explicitly involved the malicious manipulation of a software supply chain.[6]

[3] https://home.treasury.gov/system/files/136/EO-on-TikTok-8-14-20.pdf
[4] https://www.justice.gov/usao-edny/pr/china-based-executive-us-telecommunications-company-charged-disrupting-video-meetings
[5] https://www.welivesecurity.com/media_files/white-papers/Stuxnet_Under_the_Microscope.pdf
[6] https://www.fireeye.com/blog/products-and-services/2020/12/global-intrusion-campaign-leverages-software-supply-chain-compromise.html

The above events indicate that re-evaluating the security of current software distribution methods against particularly powerful adversaries is a necessary and timely task. Some academics suggest that *advanced code-signing* primitives might ameliorate the situation.[7] In this article we focus on such a primitive.

State of the art. An established approach to the authentic distribution of software updates builds on digital signature schemes. If a software version $S_i$ is updated to the next version $S_{i+1}$, the update information is signed by the original vendor so that its authenticity can be checked by the user before installing the update. This prevents malicious modifications by outsiders, including software repositories, Internet providers, etc. Unfortunately, the method provides little resilience against insider attacks in which, for instance, the vendor is coerced by a government agency to authenticate not only the legitimate $S_i \to S_{i+1}$ update, but in addition also a malicious $S_i \to S'_{i+1}$ update. If the malicious update is distributed to only a small set of high-value targets (while regular users continue to receive the legitimate version), it is unlikely that the attack is ever noticed by the victim or picked up by security researchers. The wording of the US Department of Justice, when stating that few international vendors are immune to governmental coercion attempts, suggests that such attacks are highly realistic.

Our approach. We challenge the assumption that standard signature schemes are the right tool to securely distribute software updates, and we develop a signature variant that, we argue, is better suited for the task. We start with the observation that software versions are strictly ordered (suggesting our notation $S_1 \to S_2 \to S_3 \to \ldots$), meaning that any update $S_i \to S_{i+1}$ occurs in the context of all prior updates. This sequential property is not matched by standard signature schemes which allow signing and verifying in arbitrary order. This mismatch has at least two problematic consequences: (a) Even if the signer authenticates software versions in the correct order, precautions have to be taken to avoid that a verifier accepts updates in a wrong order (e.g., $S_1 \to S_2 \to S_3 \to S_2$), for instance in the context of a software downgrade attack; and (b) The solution does not provide forward security: If the signer first authenticates $S_1 \to S_2 \to S_3$ and is then coerced to reveal its key material to an adversary, the latter can use the key material to authenticate false updates $S_1 \to S'_2$ or $S_1 \to S'_3$. A solution that provides forward security (like ours, see below) would ensure that the adversary is limited to forging on succeeding versions.

Even a signature variant that requires strictly sequential operations and provides forward security is not sufficient to protect software supply chains against state actors that coerce signers early. In the extreme case, if the signer has to reveal its keys before authenticating the very first software version, the adversary can forge on *any* (because 'following') software version. We address this by proposing a public self-enforcement mechanism: We expect of a secure solution that from any two conflicting authentications $S_i \to S_{i+1}$ and $S_i \to S'_{i+1}$ the key material that was current after authenticating $S_i$ can be recovered by invoking an efficient extraction algorithm. That is, if both $S_1 \to S_2$ and $S_1 \to S'_2$ validate correctly, then *everybody* gets into the position to also authenticate $S_1 \to S''_2$ for any $S''_2$. From the perspective of both the software vendor and the coercing state actor, getting into this situation has to be avoided by all means. (The state actor would transfer its unique privilege on to everybody else, including to competing governments, thus not only losing the priviledge but also putting its own citizens at risk.) Concretely, our mechanism (1) strongly *incentivizes a benign signer* to make its implementation as secure and robust as possible, e.g., by out-sourcing the signing operations into a Hardware Security Module; (2) strongly *disincentivizes a tempted signer* to selectively forge signatures to its own advantage; and (3) strongly *disincentivizes state actors* to coerce honest signers to reveal their keys. These properties are not met by the classic signature based software distribution method (even when enhanced by an auxiliary "detection mechanism").

**Contributions and structure.** We introduce, in Sect. 4, a new cryptographic primitive—a *sequential digital signature* (SDS) scheme—as a solution in the context of software authentication. We rigorously define suitable security properties (unforgeability with forward security, double-signing key extractability, ...), using game based models. Then, in Sect. 5, we propose generic constructions of SDS from a loosely related, simpler type of signature scheme (*strictly one-time signature*, SOT-DS, Sect. 3.2) that appeared in prior work. We implemented, tested, and evaluated our SDS scheme, and report on its efficiency in Sect. 6.

[7] https://www.scmagazine.com/perspective/encryption/can-advanced-code-signing-help-end-supply-chain-attacks

We paid special attention to ensuring that our construction can serve as a long-term solution for the secure software distribution problem. This included ensuring that it promises security also against quantum adversaries. Our SDS offers this type of security, but only if its building block, the SOT-DS scheme, does as well. Unfortunately, we found that all SOT-DS constructions proposed in prior work are based on number-theoretic assumptions like DLP and are thus *not* quantum resilient. Thus, in Sect. 5.1, as an additional contribution we construct a novel SOT-DS that is built solely from hash functions and thus represents the only known quantum-resilient candidate. We note that Germany's Federal Office for Information Security (BSI), with similar arguments, suggests prioritizing hash-based signatures for firmware authentication [4, Sect. 6.7].

**Related work.** Digital signature schemes were first proposed in the late 70's, and since then, hundreds of flavours enriched the cryptographic literature. Regular signature schemes allow the secret key owners to sign arbitrary messages in arbitrary order. In the following we discuss publications that put forward signature variants that limit this freedom and require specific relations between the signed messages to hold.

Some digital signature schemes come with self-enforcement properties. For instance, in the *e-cash* setting, the anonymous cryptocurrency users should not be allowed to spend the same coin twice by signing two different transactions. As a countermeasure, Chaum, Fiat and Naor [6] proposed a scheme that penalizes such act by automatically revealing the identity of a double-spending user.

Analogously, in a public-key infrastructure (PKI) setting, certificate authorities (CAs) generate certificates that bind keys and identities together. However, law enforcement agencies or nation states can intimidate CAs to secretly issue certificates that bind unauthentic keys to the same identities, opening the path for impersonation attacks. Poettering and Stebila proposed a self-enforcement scheme that defends against such attacks [12]. Specifically, their scheme provides the double-authentication-preventing (DAP) security property that allows certifying pairs of identities and keys, and penalizes the CA if it ever certifies different keys for the same identity. This notion was followed by later publications proposing new schemes and improving the state of the art [13,14,3,11,1,7,8].

We note that DAP signatures (from [12]) and SDS (this work) are similar in spirit in the sense that both are self-enforcing signature schemes that penalize malicious signers by leaking their key. However, SDS are strictly sequential (which fits the linearity of software updates) where DAP signatures are 'random access' (which fits the PKI application). By consequence also the security guarantees given by the two primitives are crucially different. (E.g., there cannot be forward security for DAP signatures.) Construction-wise it seems that neither are DAP signatures implied by SDS (how to remove the sequentiality?) nor are SDS implied by DAP signatures (how to add forward security?).

## 2 Notation

We write **T** and **F** for the two Boolean constants. We formalize correctness and security properties with games written in pseudo-code. The game body invokes an adversary $\mathcal{A}$ and provides it with access to zero or more oracles. A game terminates when executing a Stop with $C$ instruction, where $C$ is a Boolean expression. The truth value of $C$ is taken as the output of the game. We write $\Pr[G(\mathcal{A})]$ for the probability that game $G$ invoked with adversary $\mathcal{A}$ outputs **T**. We assume three conditional game-terminating macros: If $C$ is a Boolean expression, the game instruction Require $C$ expands to 'If not $C$: Stop with **F**', the instruction Reward $C$ expands to 'If $C$: Stop with **T**', and the instruction Promise $C$ expands to 'If not $C$: Stop with **T**'. The reader will appreciate how the macros' names correspond with their semantics in the games, where the macros are used to require a specific behaviour of the adversary, to reward the adversary for triggering a specific event, or to ensure that a specific promised property is indeed met by the scheme.

Game variables with attached brackets represent associative arrays (i.e., the dictionary data structure). For instance, the instruction '$B[4] \leftarrow 2$' assigns the value 2 to the element indexed by 4 in the array $B$, and the expression '$B[\cdot] \leftarrow 2$' initializes the entries at *all* indices to the value 2. If $X, Y$ are set variables we write $X \xleftarrow{\cup} Y$ shorthand for $X \leftarrow X \cup Y$, and if $x, y$ are vector variables we write $x \xleftarrow{\shortparallel} y$ shorthand for $x \leftarrow x \shortparallel y$, where $\shortparallel$ is the append operation. We assume the # function returns the length of a vector; for instance, if $v = (7, 8, 9)$ then $\#v = 3$.

To keep our games compact, we use the alias-creating operator $\coloneqq$ where convenient. The instruction '$A \coloneqq B$' introduces $A$ as a symbolic alias for the expression $B$. (This crucially differs from $A \leftarrow B$ which is an assignment that evaluates expression $B$ and stores the result in variable $A$.) For instance, if $B[\,]$ is an array and $B[7]$ an integer entry, and an alias is created as per $A \coloneqq B[7]$, then the instruction $A \leftarrow A + 1$ expands to $B[7] \leftarrow B[7] + 1$ and thus modifies the value of $B[7]$ (while $A$ itself is not a variable).

Some of the schemes that we formalize have stateful algorithms. A generic notation for the stateful execution of an algorithm $\alpha$ is $(\rho, y) \leftarrow \alpha(\rho, x)$ where $x$ is the algorithm's input, $y$ is the algorithm's output, and $\rho$ is the state that is updated by the execution (and thus both input and output). For compactness we use the notation $y \leftarrow \alpha\langle\rho\rangle(x)$ as a shortcut for the $(\rho, y) \leftarrow \alpha(\rho, x)$ instruction.

## 3 Stateless Signatures

We recall core principles of digital signature (DS) schemes. Our definitions are equivalent to those of prior work, but employ a slightly non-standard notation for algorithms and games. (For instance, we specify our games with an explicit verification oracle.) This will allow for an easier comparison with our definitions of stateful signatures in Sect. 4.

### 3.1 Digital Signatures: DS

Syntax. A *digital signature* (DS) scheme $S$ for a message space $\mathcal{M}$ consists of a signing key space $\mathcal{SK}$, a verification key space $\mathcal{VK}$, a signature space $\mathcal{S}$, and three efficient algorithms $gen, sig, ver$ as follows. The key generation algorithm $gen$ takes no input and outputs a signing key $sk \in \mathcal{SK}$ and a verification key $vk \in \mathcal{VK}$. The signing algorithm $sig$ is parameterized by a signing key $sk \in \mathcal{SK}$, takes a message $m \in \mathcal{M}$ on input, and outputs a signature $\sigma \in \mathcal{S}$. The verification algorithm $ver$ is parameterized by a verification key $vk \in \mathcal{VK}$, takes a message $m \in \mathcal{M}$ and a signature $\sigma \in \mathcal{S}$ on input, and outputs a Boolean value $v \in \{\mathbf{T}, \mathbf{F}\}$. Depending on whether algorithm $ver$ outputs $\mathbf{T}$ or $\mathbf{F}$ we say that it accepts or rejects. Written more compactly, a DS has the following API.

$$\begin{array}{rrrl} & gen & \xrightarrow{\text{out}} & \mathcal{SK} \times \mathcal{VK} \\ \mathcal{M} & \xrightarrow{\text{in}} & sig(\mathcal{SK}; \cdot) & \xrightarrow{\text{out}} \; \mathcal{S} \\ \mathcal{M} \times \mathcal{S} & \xrightarrow{\text{in}} & ver(\mathcal{VK}; \cdot, \cdot) & \xrightarrow{\text{out}} \; \{\mathbf{T}, \mathbf{F}\} \quad . \end{array}$$

Correctness. For $T \in \mathbb{N} \cup \{\infty\}$ we define the $T$-time correctness of a DS scheme via the game COR-DS in Fig. 1 (top), where lines 00,05,07 formalize the requirement that the signer issues at most $T$ signatures and lines 01,08,11,12 formalize the promise that all authentic signatures be accepted. Intuitively, DS scheme $S$ is *$T$-time correct* if the advantage $\mathrm{Adv}^{\text{cor-ds}}_S(\mathcal{A}) \coloneqq \Pr[\text{COR-DS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

| **Game** COR-DS($\mathcal{A}$) | **Oracle** $Sig(m)$ | **Oracle** $Ver(m, \sigma)$ |
|---|---|---|
| 00 $t \leftarrow 0$ | 05 Require $t < T$ | 10 $v \leftarrow ver(vk; m, \sigma)$ |
| 01 AC $\leftarrow \emptyset$ | 06 $\sigma \leftarrow sig(sk; m)$ | 11 If $(m, \sigma) \in$ AC: |
| 02 $(sk, vk) \leftarrow gen$ | 07 $t \leftarrow t + 1$ | 12 Promise $v$ |
| 03 $\mathcal{A}(vk)$ | 08 AC $\xleftarrow{\cup} \{(m, \sigma)\}$ | 13 Return $v$ |
| 04 Stop with $\mathbf{F}$ | 09 Return $\sigma$ | |
| **Game** SUF-DS($\mathcal{A}$) | **Oracle** $Sig(m)$ | **Oracle** $Ver(m, \sigma)$ |
| 14 $t \leftarrow 0$ | 19 Require $t < T$ | 24 $v \leftarrow ver(vk; m, \sigma)$ |
| 15 AC $\leftarrow \emptyset$ | 20 $\sigma \leftarrow sig(sk; m)$ | 25 If $(m, \sigma) \notin$ AC: |
| 16 $(sk, vk) \leftarrow gen$ | 21 $t \leftarrow t + 1$ | 26 Reward $v$ |
| 17 $\mathcal{A}(vk)$ | 22 AC $\xleftarrow{\cup} \{(m, \sigma)\}$ | 27 Return $v$ |
| 18 Stop with $\mathbf{F}$ | 23 Return $\sigma$ | |

**Fig. 1.** Games COR-DS and SUF-DS for defining the correctness and strong unforgeability, respectively, of a DS scheme. Set AC indicates the authentic message-signature pairs.

UNFORGEABILITY. With respect to security we define what it means for a DS scheme to provide ($T$-time) strong unforgeability. A scheme meets this property if all valid signatures that an adversary can come up with are replays of signatures priorly generated by the signer. This is formalized via the game SUF-DS in Fig. 1 (bottom), where lines 15,22,25,26 reward the adversary if it delivers a message-signature pair that is accepted despite being non-authentic. Intuitively, DS scheme $S$ is *$T$-time strongly unforgeable* if the advantage $\mathrm{Adv}^{\text{suf-ds}}_{S}(\mathcal{A}) := \Pr[\text{SUF-DS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

### 3.2 Strictly one-time digital signatures: SOT-DS

If more than $T$ messages are signed with a $T$-time signature scheme instance, then the instance is not anymore guaranteed to be secure. However, this does not imply that the instance's security collapses completely, i.e., that forging signatures on arbitrary messages suddenly becomes easy. In the following we recall the definition of a stronger form of DS that for the special case of one-time signatures guarantees that signing twice unavoidably leads to a maximum loss of security: A *strictly one-time digital signature* (SOT-DS, [11]) scheme is a 1-time DS scheme where any two signatures created with the same signing key immediately fully expose that signing key and thus enable universal forging. Note that [11] provides SOT-DS constructions that leverage on zero-knowledge proof systems over number-theoretic assumptions (like DLP). In Sect. 5.1 we specify two novel constructions that are based on hash functions.

SYNTAX. A SOT-DS scheme [11, Sect. 3.2] consists of the algorithms *gen*, *sig*, *ver* of a regular DS scheme, plus an additional key extraction algorithm that recovers the signing key from any two valid signatures. Precisely, the *ext* algorithm is parameterized by a verification key $vk \in \mathcal{VK}$, takes two message-signature pairs $(m^0, \sigma^0), (m^1, \sigma^1) \in \mathcal{M} \times \mathcal{S}$ on input, and outputs a signing key $sk \in \mathcal{SK}$. More compactly,

$$(\mathcal{M} \times \mathcal{S}) \times (\mathcal{M} \times \mathcal{S}) \xrightarrow{\text{in}} ext(\mathcal{VK}; \cdot, \cdot) \xrightarrow{\text{out}} \mathcal{SK} \ .$$

EXTRACTABILITY. The key extractability feature of a SOT-DS is formalized via the game KEX-DS in Fig. 2. In the game, a malicious signer that outputs two message-signature pairs (line 01) such that both signatures are valid (lines 02,03) yet the message-signature pairs are different (line 04) is rewarded if the key extraction (line 05) recovers a wrong key (line 06). Intuitively, SOT-DS scheme $S$ is *(one-time) key extractable* if the advantage $\mathrm{Adv}^{\text{kex-ds}}_{S}(\mathcal{A}) := \Pr[\text{KEX-DS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

**Game** KEX-DS($\mathcal{A}$)
00 $(sk, vk) \leftarrow gen$
01 $(m^0, \sigma^0), (m^1, \sigma^1) \leftarrow \mathcal{A}(sk, vk)$
02 Require $ver(vk; m^0, \sigma^0) = \mathbf{T}$
03 Require $ver(vk; m^1, \sigma^1) = \mathbf{T}$
04 Require $(m^0, \sigma^0) \neq (m^1, \sigma^1)$
05 $sk^* \leftarrow ext(vk; m^0, \sigma^0, m^1, \sigma^1)$
06 Reward $sk^* \neq sk$
07 Stop with **F**

**Fig. 2.** Game KEX-DS for defining the key extractability of a SOT-DS scheme. Adversary $\mathcal{A}$ takes the role of a malicious signer and thus, in line 01, receives direct access to the signing key. (A consequence of this is that the game doesn't need to provide a signing oracle.)

## 4 Sequential Digital Signatures: SDS

A sequential digital signature (SDS) is a variant of a regular digital signature (DS) where both the signer and the verifier are stateful. In SDS, message-signature pairs have to be verified in the same order as they are generated. This restriction of functionality is paired with a strengthening of security, and we argue that the latter makes the option of replacing a DS by an SDS, in applications where this is possible, attractive. In particular, we argue that using an SDS is advantageous over using a DS for the purpose

of authenticating program code. This is both because an SDS provides forward security (i.e., maintains, after a signing key corruption, as much security as possible) and because an SDS may come with a self-enforcement mechanism that penalizes a signer that offends the rule of signing strictly sequentially.

In the following we formalize the SDS notion. We took efforts to align the syntax of DS and SDS as much as possible, and to also let the security notions and games correspond to each other. (This explains why in Sect. 3 we decided to use the slightly non-standard notation.)

Syntax. A *sequential digital signature* (SDS) scheme $S$ for a message space $\mathcal{M}$ consists of a signing state space $\mathcal{SST}$, a verification state space $\mathcal{VST}$, a signature space $\mathcal{S}$, and three efficient algorithms $gen, sig, ver$ as follows. The initialization algorithm $gen$ takes no input and outputs an initial signing state $sst \in \mathcal{SST}$ and an initial verification state $vst \in \mathcal{VST}$. The signing algorithm $sig$ depends on a signing state $sst \in \mathcal{SST}$ which it may update, takes a message $m \in \mathcal{M}$ on input, and outputs a signature $\sigma \in \mathcal{S}$. The verification algorithm $ver$ depends on a verification state $vst \in \mathcal{VST}$ which it may update, takes a message $m \in \mathcal{M}$ and a signature $\sigma \in \mathcal{S}$ on input, and outputs a Boolean value $v \in \{\mathbf{T}, \mathbf{F}\}$. Depending on whether algorithm $ver$ outputs $\mathbf{T}$ or $\mathbf{F}$ we say that it accepts or rejects. More compactly, using the state-update notation from Sect. 2,

$$\begin{array}{rcl} & gen & \xrightarrow{\text{out}} \mathcal{SST} \times \mathcal{VST} \\ \mathcal{M} \xrightarrow{\text{in}} & sig\langle \mathcal{SST} \rangle(\cdot) & \xrightarrow{\text{out}} \mathcal{S} \\ \mathcal{M} \times \mathcal{S} \xrightarrow{\text{in}} & ver\langle \mathcal{VST} \rangle(\cdot,\cdot) & \xrightarrow{\text{out}} \{\mathbf{T}, \mathbf{F}\} \quad . \end{array}$$

Before proceeding with defining the correctness and security properties of SDS, we introduce the notions of signing history and verification history, explain why useful security definitions for SDS have to consider the existence of multiple independent verifiers, and indicate how our models capture forward security.

Signing/Verification History. Once an initial SDS signing state $sst$ was created, a series $m_1, m_2, \ldots$ of messages can be authenticated by iteratively invoking $\sigma_i \leftarrow sig\langle sst \rangle(m_i)$ (where each invocation of $sig$ may update $sst$). By the sequentiality of this process, the signing state $sst$ can be assumed to always reflect, explicitly or implicitly, the *signing history* $(m_1, \sigma_1) \shortparallel (m_2, \sigma_2) \shortparallel \ldots$ processed so far. Similarly, a *verification history* consists of the messages and signatures that a verifier accepted as authentic when iteratively invoked as per $v_i \leftarrow ver\langle vst \rangle(m_i, \sigma_i)$. (If $ver$ is invoked with a message-signature pair that is rejected, this pair is *not* recorded in the verification history.[8]) In our SDS games, the *Sig* and *Ver* oracles record signing and verification histories by appending generated or accepted message-signature pairs to game variables $sh$ and $vh$, respectively.

Multiple Verifiers. A real-world characteristic of (stateless or stateful) signature schemes is that signers are typically matched by multiple independent verifiers that check their signatures. However, as in the stateless case there is no component that could individualize different verifiers (all verifiers receive the same input and cannot memorize anything), when formalizing the correctness and security of such schemes it is sufficient, without loss of generality, to consider a single verifier. The same is not possible for SDS where verifiers that are exposed to different sequences of valid or invalid message-signature pairs may end up in different states.[9] Our games hence explicitly model a setup where a single signer is matched by an unlimited number of independent verifiers. This is implemented by requiring the adversary to explicitly indicate, for each of its *Ver* queries, the identifier $id$ of the verifier instance that is meant to process the provided message-signature pair. We use the array variable VST[·] (see Sect. 2) to store the states of all verifiers so that its entry VST[$id$] represents the state of the verifier with identifier $id$. Similarly our games use the variable VH[·] to store the verification histories of all verifiers so that entry VH[$id$] represents the verification history of verifier $id$. Note that the values of $id$ are chosen at the discretion of the adversary. The identifiers are only used to implement the game logics while the SDS algorithms themselves will never learn them.

[8] This does not preclude that information about the rejected message-signature pair is reflected in $vst$ which can be updated even if the verification fails.

[9] If an SDS verification algorithm is randomized then the verification states of different verifiers might diverge even when provided with the same sequence of message-signature pairs.

FORWARD SECURITY. Our models capture the forward security aspect by giving the adversary the option to *corrupt* the signer by invoking a dedicated oracle that returns a copy of the signer's current state.[10] Intuitively, before such a corruption happens, exclusively the signer is able to create valid signatures, i.e., is the only *authoritative* party. However, by invoking the *Corrupt* oracle also the adversary gets into the position to craft valid signatures (by applying the regular signing algorithm to the retrieved state), making the signer lose its authoritativeness. The notion of authoritativeness will appear explicitly in our security games.

**Correctness of SDS.** For $T \in \mathbb{N} \cup \{\infty\}$ we define the $T$-time correctness of an SDS scheme via the game COR-SDS in Fig. 3, where lines 00,09,11 formalize the requirement that the signer issues at most $T$ signatures and lines 01,13,19,20 formalize the promise that all authentic signing histories be accepted. Intuitively, SDS scheme $S$ is *$T$-time correct* if the advantage $\mathrm{Adv}^{\text{cor-sds}}_{S}(\mathcal{A}) := \Pr[\text{COR-SDS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

| **Game** COR-SDS($\mathcal{A}$) | **Oracle** $Sig(m)$ | **Oracle** $Ver(id, m, \sigma)$ |
|---|---|---|
| 00 $t \leftarrow 0$ | 09 Require $t < T$ | 16 $vh := \mathrm{VH}[id]$ |
| 01 $\mathrm{AC} \leftarrow \emptyset$ | 10 $\sigma \leftarrow sig\langle sst\rangle(m)$ | 17 $vst := \mathrm{VST}[id]$ |
| 02 $sh \leftarrow \epsilon$ | 11 $t \leftarrow t + 1$ | 18 $v \leftarrow ver\langle vst\rangle(m, \sigma)$ |
| 03 $\mathrm{VH}[\cdot] \leftarrow \epsilon$ | 12 $sh \xleftarrow{\shortparallel} (m, \sigma)$ | 19 If $vh \shortparallel (m, \sigma) \in \mathrm{AC}$: |
| 04 $(sst_0, vst_0) \leftarrow gen$ | 13 $\mathrm{AC} \xleftarrow{\cup} \{sh\}$ | 20 Promise $v$ |
| 05 $sst \leftarrow sst_0$ | 14 Return $\sigma$ | 21 If $v$: $vh \xleftarrow{\shortparallel} (m, \sigma)$ |
| 06 $\mathrm{VST}[\cdot] \leftarrow vst_0$ | **Oracle** *Corrupt* | 22 Return $v, vst$ |
| 07 $\mathcal{A}(vst_0)$ | 15 Return $sst$ | |
| 08 Stop with **F** | | |

**Fig. 3.** Game COR-SDS for defining the correctness of an SDS scheme. Set AC indicates the authentic signing histories. Vector $sh$ indicates the (signer's) signing history. For any verifier identity $id$, vector VH[$id$] indicates that verifier's verification history and VST[$id$] represents its verification state. (Recall from Sect. 2 that the instructions in lines 03,06 assign the same initial value to all array entries while the instructions in lines 16,17 create symbolic aliases.)

**Unforgeability of SDS.** Our security models capture forward security. Corrupting a signer unavoidably brings the adversary into the position to forge signatures on messages associated with points in time following the corruption, but if a scheme is forward secure then signatures for points in time preceding the corruption remain unforgeable. We start with making the concepts of the past and the future of a signer more precise.

In the correctness game of Fig. 3, a signing history is represented by a string $sh \in \mathcal{X}^*$ over the alphabet $\mathcal{X} = \mathcal{M} \times \mathcal{S}$ which is the universe of all message-signature pairs. Given a signing history $sh$, its past $\mathrm{Past}(sh)$ consists of the signing histories that it developed from, and its future $\mathrm{Future}(sh)$ consists of the signing histories that it could develop into. Formally, if $\prec$ denotes the (anti-reflexive) is-prefix-of relation on strings in $\mathcal{X}^*$, we let:

$$\mathrm{Past}(sh) = \{sh' \in \mathcal{X}^* : sh' \prec sh\}$$
$$\mathrm{Future}(sh) = \{sh' \in \mathcal{X}^* : sh \prec sh'\}$$

Note that we have $\mathrm{Past}(\epsilon) = \emptyset$ and $\mathrm{Future}(\epsilon) = \mathcal{X}^+$. ("Not having a past means having all options for the future.")

Recalling the concept of authoritativeness discussed above, let us now observe that (1) initially, until a signer is corrupted, it is authoritative for its entire future; and (2) corrupting a signer means that its authoritativeness is revoked for what follows the corruption. In our security games for SDS, game variable AE indicates for which signing histories the signer is authoritative. We implement the two observations by (1) starting the games with $\mathrm{AE} = \mathrm{Future}(\epsilon) = \mathcal{X}^+$; and (2) letting $\mathrm{AE} \leftarrow \mathrm{AE} \setminus \mathrm{Future}(sh)$ whenever a corruption query is posed.

[10] Our verification oracles return copies of the verifier states right away, fully removing the need of having to think about also adding a corruption oracle for verifiers.

We are now ready to state our unforgeability definition for SDS schemes. For $T \in \mathbb{N} \cup \{\infty\}$ we define the $T$-time strong unforgeability of an SDS scheme via the game SUF-SDS in Fig. 4, where lines 02,16 implement the tracking of the signer's authoritativeness and lines 01,14,21,22 reward the adversary if it makes a verifier accept a non-authentic verification history, and this happens for a point where the signer should be authoritative. Intuitively, SDS scheme $S$ is *$T$-time strongly unforgeable* (with forward security) if the advantage $\mathrm{Adv}_S^{\text{suf-sds}}(\mathcal{A}) := \Pr[\text{SUF-SDS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

| **Game** SUF-SDS($\mathcal{A}$) | **Oracle** $Sig(m)$ | **Oracle** $Ver(id, m, \sigma)$ |
|---|---|---|
| 00 $t \leftarrow 0$ | 10 Require $t < T$ | 18 $vh := \mathrm{VH}[id]$ |
| 01 $\mathrm{AC} \leftarrow \emptyset$ | 11 $\sigma \leftarrow sig\langle sst\rangle(m)$ | 19 $vst := \mathrm{VST}[id]$ |
| 02 $\mathrm{AE} \leftarrow \mathrm{Future}(\epsilon)$ | 12 $t \leftarrow t + 1$ | 20 $v \leftarrow ver\langle vst\rangle(m, \sigma)$ |
| 03 $sh \leftarrow \epsilon$ | 13 $sh \xleftarrow{\shortparallel} (m, \sigma)$ | 21 If $vh \shortparallel (m, \sigma) \in \mathrm{AE} \setminus \mathrm{AC}$: |
| 04 $\mathrm{VH}[\cdot] \leftarrow \epsilon$ | 14 $\mathrm{AC} \xleftarrow{\cup} \{sh\}$ | 22 Reward $v$ |
| 05 $(sst_0, vst_0) \leftarrow gen$ | 15 Return $\sigma$ | 23 If $v$: $vh \xleftarrow{\shortparallel} (m, \sigma)$ |
| 06 $sst \leftarrow sst_0$ | | 24 Return $v, vst$ |
| 07 $\mathrm{VST}[\cdot] \leftarrow vst_0$ | **Oracle** *Corrupt* | |
| 08 $\mathcal{A}(vst_0)$ | 16 $\mathrm{AE} \leftarrow \mathrm{AE} \setminus \mathrm{Future}(sh)$ | |
| 09 Stop with **F** | 17 Return $sst$ | |

**Fig. 4.** Game SUF-SDS for defining the strong unforgeability of an SDS scheme. Set AE indicates the signing histories for which the signer is authoritative. See the caption of Fig. 3 for the meaning of other variables and symbols.

**Extractability of SDS.** We formalize two self-enforcement properties for SDS: double-signature forgeability and double-signature extractability. Observe that, assuming regular operations, for the verification histories $vh_1, vh_2$ of any two verifiers we have either $vh_1 \preceq vh_2$ or $vh_2 \preceq vh_1$, that is, the verification histories are identical modulo one of them possibly lagging behind. We refer to verification histories that don't follow this pattern as 'conflicting'.

**Definition 1.** *Two verification histories $vh_1, vh_2 \in \mathcal{X}^+$ are* conflicting *if they diverge, i.e., if $vh_1 = vh' \shortparallel P_1 \shortparallel vh''$ and $vh_2 = vh' \shortparallel P_2 \shortparallel vh'''$ where $vh' \in \mathcal{X}^*$ is a (possibly empty) common prefix, $P_1 = (m_1, \sigma_1) \in \mathcal{X}$ and $P_2 = (m_2, \sigma_2) \in \mathcal{X}$ are different message-signature pairs, and $vh'', vh''' \in \mathcal{X}^*$ are arbitrary (possibly empty) suffixes.*

In the following we consider irregular operations, i.e., the case where conflicting verification histories do emerge in the games. Reasons for conflicting histories include that the signer is not honest or is impersonated after being corrupted. Our aim is to disincentivize irregular behavior as much as possible, and we do this by demanding that from any conflicting pair of verification histories it be possible to forge signatures on arbitrary (future) messages with the same ease as if the forger knew the correct keys.

The security goal of double-signature forgeability (DSF) demands that from any conflicting pair of verification histories one can forge signatures using a dedicated algorithm *forge* with the following API:

$$\mathcal{VST} \times \mathcal{X}^* \times \mathcal{X} \times \mathcal{X} \times \mathcal{M}^+ \xrightarrow{\text{in}} forge \xrightarrow{\text{out}} \mathcal{S}^+ \ .$$

Using the notation from the above definition, if *forge* is invoked as $\bar{\sigma} \leftarrow forge(vst_0, vh', P_1, P_2, \bar{m}_1 \shortparallel \ldots \shortparallel \bar{m}_l)$, where $vst_0$ is the initial verification key, then the signatures in $\bar{\sigma} = \bar{\sigma}_1 \shortparallel \ldots \shortparallel \bar{\sigma}_l$ shall be valid in the sense that $vh^* = vh' \shortparallel (\bar{m}_1, \bar{\sigma}_1) \shortparallel \ldots \shortparallel (\bar{m}_l, \bar{\sigma}_l)$ is a verification history accepted by any verifier.

For $T \in \mathbb{N} \cup \{\infty\}$ we define the $T$-time double-signature forgeability of an SDS scheme via the game DSF-SDS in Fig. 5, where lines 00–08 prepare the inputs for the *forge* algorithm and lines 10–15 declare the *forge*-crafted signatures to be as authentic as real ones by adding them to the set AC (so that they can be tested in the *Ver* oracle). Intuitively, SDS scheme $S$ is *$T$-time double-signature forgeable* if the advantage $\mathrm{Adv}_S^{\text{dsf-sds}}(\mathcal{A}) := \Pr[\text{DSF-SDS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

Our second security goal of double-signature extractability (DSE) is strictly stronger than DSF and demands that from any conflicting pair of verification histories the signing state that was current when

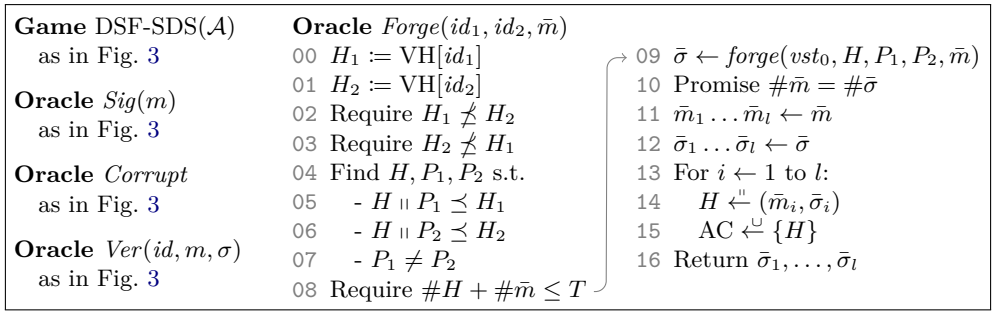


**Fig. 5.** Game DSF-SDS for defining the double-signature forgeability of an SDS scheme. See also the caption of Fig. 3 for the meaning of game variables.

the double-signing happened can be extracted. Specifically, we demand that an extraction algorithm *ext* exists with the following API:

$$\mathcal{VST} \times \mathcal{X}^* \times \mathcal{X} \times \mathcal{X} \xrightarrow{\text{in}} ext \xrightarrow{\text{out}} \mathcal{SST} \ .$$

For instance, if in the above example we have $vh' = \epsilon$ then invoking $sst' \leftarrow ext(vst_0, vh', P_1, P_2)$ shall extract $sst' = sst_0$. It is easy to see that extractability implies forgeability. Demanding that *the* signing state is extracted makes sense only if there is at most one possible candidate. While not every SDS scheme meets this requirement (e.g., if the *sig* algorithm is randomized), our constructions from Sect. 5 do. We will thus analyze them in the stronger model (DSE).

For $T \in \mathbb{N} \cup \{\infty\}$ we define the $T$-time double-signature extractability of an SDS scheme via the game DSE-SDS in Fig. 6, where lines 00–08 prepare the inputs for the *ext* algorithm and lines 10–14 test that the extracted state is correct. Intuitively, SDS scheme $S$ is *$T$-time double-signature extractable* if the advantage $\mathrm{Adv}_S^{\text{dse-sds}}(\mathcal{A}) := \Pr[\text{DSE-SDS}(\mathcal{A})]$ is negligible for all efficient adversaries $\mathcal{A}$.

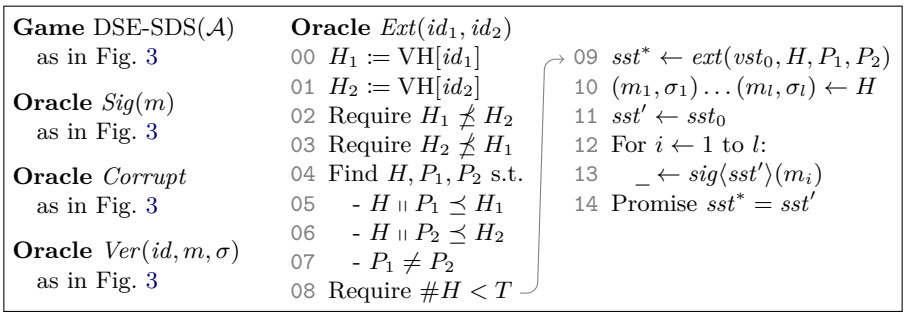


**Fig. 6.** Game DSE-SDS for defining the double-signature extractability of an SDS scheme. The notation in line 13 means that the output of *sig* is ignored. See also the caption of Fig. 3 for the meaning of game variables.

## 5 Constructions

We construct an SDS that provably fulfills the security properties defined in Sect. 4. The scheme leverages on a strictly one-time digital signature scheme (SOT-DS, see Sect. 3.2) as a building block. Prior work [11] succeeded with constructing SOT-DS, and any of these constructions can be used in our context. However, as they are based on number-theoretic assumptions like the DLP, which are known not to withstand quantum adversaries, we also propose two variants of a novel SOT-DS construction that is based solely on hash functions.

### 5.1 Hash function based SOT-DS

Recall the classic hash function based (one-time) signature scheme by Lamport [10] where the signing key is a matrix $(x_i^b)_{b\in\{0,1\},1\le i\le n}$ of randomly picked hash function pre-images $x_i^b$, the verification key is the matrix $(y_i^b)_{b\in\{0,1\},1\le i\le n}$ of hash function images $y_i^b = H(x_i^b)$, and signing the $n$-bit message $m = m_1 \dots m_n$ corresponds with releasing the pre-images $(x_i^{m_i})_{1\le i\le n}$ as the signature. Recall further from Sect. 3.2 that a SOT-DS scheme is a one-time signature scheme where double-signing, i.e., signing two different messages with the same key, implies losing the signing key to the public by means of a dedicated *ext* algorithm. While Lamport's scheme releases key components as part of the signing process, and double-signing revokes its existential unforgeability property, double-signing does not release enough information to create signatures on arbitrary messages. The scheme is thus not a SOT-DS instance.

We propose to turn Lamport's scheme into a SOT-DS as follows. The first step is to generate the $b = 0$ half of the pre-images $x_i^b$ deterministically from a seed $k$ using a pseudo-random generator (PRG), i.e., we let $(x_1^0, \dots, x_n^0) \leftarrow G(k)$.[11] Instead of choosing the $b = 1$ half uniformly distributed as well (either by random assignment or by using a PRG), we assign the $x_i^1$ values such that any pair $(x_i^0, x_i^1)$ allows extracting the seed $k$. Concretely we let $x_i^1 \leftarrow k \oplus x_i^0$ for all $i$, where $\oplus$ denotes XOR. Double-signing in Lamport's scheme corresponds with releasing, for at least one index $i$, the preimage $x_i^0$ in the one signature and the preimage $x_i^1$ in the other signature. That is, if we consider $k$ the signing key of the scheme, the extractability goal is attained. We still need to confirm that the scheme also provides unforgeability, but an analysis in the random oracle model for $G, H$ will show that this is indeed the case.

We derive two different yet closely-related SOT-DS constructions from the above intuition. The difference between the schemes is how verification keys and signatures are represented. The sizes of signing keys, verification keys, and signatures are 1, $2n$, and $n$ elements, respectively, for our first SOT-DS scheme, and are 1, 1, $2n$, respectively, for the second. That is, the latter has considerably smaller verification keys at the cost of a doubled signature length.

| **Algo** *gen* | **Algo** *sig*$(sk; m)$ | **Algo** *ver*$(vk; m, \sigma)$ | **Algo** *ext*$(vk; m^0, \sigma^0, m^1, \sigma^1)$ |
|---|---|---|---|
| 00 $k \leftarrow \$(\{0,1\}^l)$ | 09 $m_1, \dots, m_n \leftarrow m$ | 17 $m_1, \dots, m_n \leftarrow m$ | 23 Require $m^0 \neq m^1$ |
| 01 $(sk_1^0, \dots, sk_n^0) \leftarrow G(k)$ | 10 $(sk_1^0, \dots, sk_n^0) \leftarrow G(k)$ | 18 For $i \in \{1, \dots, n\}$: | 24 $m_1^0, \dots, m_n^0 \leftarrow m^0$ |
| 02 For $i \in \{1, \dots, n\}$: | 11 For $i \in \{1, \dots, n\}$: | 19 $b \leftarrow m_i$ | 25 $m_1^1, \dots, m_n^1 \leftarrow m^1$ |
| 03 $sk_i^1 \leftarrow k \oplus sk_i^0$ | 12 $sk_i^1 \leftarrow k \oplus sk_i^0$ | 20 If $vk_i^b \neq H_i^b(\sigma_i)$: | 26 Find $i$ s.t. $m_i^0 \neq m_i^1$ |
| 04 For $b \in \{0,1\}$: | 13 $b \leftarrow m_i$ | 21 Reject | 27 W.l.o.g. $m_i^0 = 0 \wedge m_i^1 = 1$ |
| 05 $vk_i^b \leftarrow H_i^b(sk_i^b)$ | 14 $\sigma_i \leftarrow sk_i^b$ | 22 Accept | 28 $sk_i^0 \leftarrow \sigma_i^0$; $sk_i^1 \leftarrow \sigma_i^1$ |
| 06 $sk \coloneqq k$ | 15 $\sigma \coloneqq (\sigma_i)_i$ | | 29 $k \leftarrow sk_i^1 \oplus sk_i^0$ |
| 07 $vk \coloneqq (vk_i^b)_{b,i}$ | 16 Return $\sigma$ | | 30 $sk \coloneqq k$ |
| 08 Return $sk, vk$ | | | 31 Return $sk$ |

**Fig. 7.** First SOT-DS construction. We write $k \leftarrow \$(\{0,1\}^l)$ for randomly sampling an $l$-bit seed $k$. We write $H_i^b(x)$ shorthand for $H(b, i, x)$. We write Accept for Return **T** and Reject for Return **F**. Algorithm *ext* assumes for its inputs that $ver(vk; m^0, \sigma^0) = \mathbf{T}$ and $ver(vk; m^1, \sigma^1) = \mathbf{T}$ and $(m^0, \sigma^0) \neq (m^1, \sigma^1)$, as it is promised by lines 02,03,04 of Fig. 2. As the third condition effectively implies $m^0 \neq m^1$ if $H$ is collision resistant, the algorithm will not abort in line 23 and the instruction of line 26 is guaranteed to succeed.

Details of first construction. This scheme is defined for a message space $\mathcal{M} = \{0,1\}^n$ and a security parameter $l \in \mathbb{N}$ (think of $l = 256$). Let $\mathcal{SK} = \{0,1\}^l$ and $\mathcal{VK} = \{0,1\}^{2\times n\times l}$ and $\mathcal{S} = \{0,1\}^{n\times l}$. Let $G\colon \{0,1\}^l \to \{0,1\}^{n\times l}$ be a PRG and $H\colon \{0,1\} \times \{1, \dots, n\} \times \{0,1\}^l \to \{0,1\}^l$ a hash function. Both $G$ and $H$ will be modeled as random oracles and can be easily constructed from, say, SHA256. Our SOT-DS algorithms *gen*, *sig*, *ver*, *ext* are then defined as in Fig. 7.

It is straightforward to verify that the scheme provides correctness and extractability by the definitions of Sect. 3. With respect to (one-time, strong) unforgeability, observe that all employed cryptographic primitives are random oracles, that is, the security argument will be combinatoric in nature. Specifically, note that any strong forgery $(m^*, \sigma^*)$ that the adversary can come up with includes at least one fresh pre-image $x$ such that $H_i^b(x) = vk_i^b$, for some $b, i$. Finding such a pre-image from the random oracle alone

[11] Other hash-based signature schemes like SPHINCS$^+$ use similar techniques [2,5].

is effectively infeasible: each attempt succeeds with probability $2^{-l}$. The other approach would involve guessing value $k$ (given just random oracle images), and again this succeeds only with probability $2^{-l}$ per attempt. Even if the adversary makes polynomially many queries to the random oracles, the resulting security bound will be of the type $p/2^{-l}$, for a polynomial $p$, which is negligible.

Details of second construction. Our second construction is very much like the first but it has $\mathcal{VK} = \{0,1\}^l$ and $\mathcal{S} = \{0,1\}^{2\times n\times l}$. It is based on the observation that the $sk_i^b$ components included in the signatures of Fig. 7 allow the verifier to recover the $vk_i^b$ components of the verification key. The idea is thus to replace the verification key $vk$ by a value $H^{\#}(vk)$, where $H^{\#}\colon \{0,1\}^{2\times n\times l} \to \{0,1\}^l$ is an auxiliary collision resistant hash function, to include in each signature the verification key components missing to recover the complete verification key, and to then recover the key and verify it based on the present hash value. The algorithms of our scheme appear in Fig. 8. The security arguments are analogues of the ones given above, with the collision resistance of $H^{\#}$ added to the list of assumptions.

| **Algo** $gen$ | **Algo** $sig(sk;m)$ | **Algo** $ver(vk;m,\sigma)$ | **Algo** $ext(vk;m^0,\sigma^0,m^1,\sigma^1)$ |
|---|---|---|---|
| 00 $k \leftarrow \$(\{0,1\}^l)$ | 10 $m_1,\ldots,m_n \leftarrow m$ | 20 $m_1,\ldots,m_n \leftarrow m$ | 30 Require $m^0 \neq m^1$ |
| 01 $(sk_1^0,\ldots,sk_n^0) \leftarrow G(k)$ | 11 $(sk_1^0,\ldots,sk_n^0) \leftarrow G(k)$ | 21 For $i \in \{1,\ldots,n\}$: | 31 $m_1^0,\ldots,m_n^0 \leftarrow m^0$ |
| 02 For $i \in \{1,\ldots,n\}$: | 12 For $i \in \{1,\ldots,n\}$: | 22 $b \leftarrow m_i$ | 32 $m_1^1,\ldots,m_n^1 \leftarrow m^1$ |
| 03 $sk_i^1 \leftarrow k \oplus sk_i^0$ | 13 $sk_i^1 \leftarrow k \oplus sk_i^0$ | 23 $d \leftarrow 1 - m_i$ | 33 Find $i$ s.t. $m_i^0 \neq m_i^1$ |
| 04 For $b \in \{0,1\}$: | 14 $b \leftarrow m_i$ | 24 $(sk_i^b, vk_i^d) \leftarrow \sigma_i$ | 34 $(sk_i^0,\_) \leftarrow \sigma_i^0$ |
| 05 $vk_i^b \leftarrow H_i^b(sk_i^b)$ | 15 $d \leftarrow 1 - m_i$ | 25 $vk_i^b \leftarrow H_i^b(sk_i^b)$ | 35 $(sk_i^1,\_) \leftarrow \sigma_i^1$ |
| 06 $vk' \leftarrow (vk_i^b)_{b,i}$ | 16 $vk_i^d \leftarrow H_i^d(sk_i^d)$ | 26 $vk' \leftarrow (vk_i^b)_{b,i}$ | 36 $k \leftarrow sk_i^0 \oplus sk_i^1$ |
| 07 $sk \coloneqq k$ | 17 $\sigma_i \leftarrow (sk_i^b, vk_i^d)$ | 27 If $vk \neq H^{\#}(vk')$: | 37 $sk \coloneqq k$ |
| 08 $vk \leftarrow H^{\#}(vk')$ | 18 $\sigma \coloneqq (\sigma_i)_i$ | 28 Reject | 38 Return $sk$ |
| 09 Return $sk, vk$ | 19 Return $\sigma$ | 29 Accept | |

**Fig. 8.** Second SOT-DS construction. We denote with $H^{\#}$ an auxiliary collision-resistant hash function. See also the caption of Fig. 7.

### 5.2 SDS from SOT-DS

We use a SOT-DS scheme as a building block to construct a sequential digital signature (SDS) scheme that fulfills the security properties defined in Sect. 4. In the following we use the notation $\overline{gen}$, $\overline{sig}$, $\overline{ver}$, $\overline{ext}$ for SOT-DS algorithms and $gen$, $sig$, $ver$, $ext$ for SDS algorithms. For $\overline{gen}$ we assume that its randomness space is $\{0,1\}^l$ for some $l \in \mathbb{N}$. Our construction works by chaining multiple SOT-DS instances together. It supports a maximum of $T$ periods for a configurable $T \in \mathbb{N}$ and generates a total of $T$ SOT-DS instances as follows: While the first SOT-DS is generated regularly by invoking $\overline{gen}$ with fresh randomness, the remaining $T-1$ instances are generated by invoking $\overline{gen}$ with explicitly specified randomness that is derived with a random oracle $H\colon \bar{\mathcal{S}} \to \{0,1\}^l$ from the preceeding SOT-DS signing key: If $(sk_i, vk_i)$ is the SOT-DS key pair of the $i$-th epoch, then the SOT-DS key pair $(sk_{i+1}, vk_{i+1})$ of the next epoch is derived by letting $k \leftarrow H(sk_i)$ and $(sk_{i+1}, vk_{i+1}) \leftarrow \overline{gen}[k]$, where the bracket notation means the algorithm uses the explicitly specified randomness $k$.[12] The SDS signing state of the first epoch is $(sk_1, 1)$. The signing state deterministically evolves by one position after each signing operation. To achieve forward security, switching to the next SDS epoch also involves securely erasing the old SOT-DS signing key. The SDS verification state is the vector of all SOT-DS verification keys, plus an indication of the current epoch. (We also clear verification state elements that become redundant over time, but this is for efficiency and not for security.) The explicit specification of the scheme algorithms is in Fig. 9.

The correctness of the SDS scheme follows immediately from the correctness of the SOT-DS scheme. The (strong) unforgeability is easily reduced to the SOT-DS unforgeability: If the verifier's initial state is authentic, then any SDS forgery immediately translates to an SOT-DS forgery *or* the evaluation of the random oracle $H$ on input the previous signing key (in which case the reduction goes to the unforgeability of the previous SOT-DS instance). By the strict sequentiality of the construction, the forward security

[12] The random oracle $H$ used here should of course be independent of the random oracle with the same name of Sect. 5.1.

| **Algo** $gen_T$ | **Algo** $sig_T\langle sst\rangle(m)$ | **Algo** $ver_T\langle vst\rangle(m,\sigma)$ | **Algo** $ext_T(vst_0; vh', P_1, P_2)$ |
|---|---|---|---|
| 00 $k \leftarrow \$(\{0,1\}^l)$ | 08 $(sk_t, t) \leftarrow sst$ | 16 $(vk, t) \leftarrow vst$ | 23 $(vk, 1) \leftarrow vst_0$ |
| 01 For $t \leftarrow 1$ to $T$: | 09 Require $1 \le t \le T$ | 17 Require $1 \le t \le T$ | 24 $t \leftarrow \#vh' + 1$ |
| 02 $\quad (sk_t, vk_t) \leftarrow \overline{gen}[k]$ | 10 $\sigma \leftarrow \overline{sig}(sk_t; m)$ | 18 $v \leftarrow \overline{ver}(vk_t; m, \sigma)$ | 25 $sk_t \leftarrow \overline{ext}(vk_t; P_1, P_2)$ |
| 03 $\quad k \leftarrow H(sk_t)$ | 11 $k \leftarrow H(sk_t)$ | 19 Require $v$ | 26 $sst \leftarrow (sk_t, t)$ |
| 04 $sst \leftarrow (sk_1, 1)$ | 12 Securely erase $sk_t$ | 20 $vk_t \leftarrow \perp$ | 27 Return $sst$ |
| 05 $vk \coloneqq (vk_1, \ldots, vk_T)$ | 13 $(sk_{t+1}, \_) \leftarrow \overline{gen}[k]$ | 21 $vst \leftarrow (vk, t+1)$ | |
| 06 $vst \leftarrow (vk, 1)$ | 14 $sst \leftarrow (sk_{t+1}, t+1)$ | 22 Return $v$ | |
| 07 Return $(sst, vst)$ | 15 Return $\sigma$ | | |

**Fig. 9.** SDS construction. Parameter $T \in \mathbb{N}$ indicates the number of supported epochs and can be fixed arbitrarily. In line 02, $\overline{gen}$ is invoked with explicit randomness. Extraction algorithm $ext$ assumes for its inputs $P_1, P_2$ what it is promised by the DSE game in Fig. 6.

basically comes for free. Let's finally consider the extractability property. Two conflicting SDS verification histories translate immediately to two conflicting SOT-DS message-signature pairs, allowing for the recovery of the SOT-DS signing key of that epoch. The latter is precisely the SDS signing state that is to be recovered.

## 6 Implementation and evaluation

In order to experimentally evaluate our SDS construction from Sect. 5.2, we implemented its algorithms in the C programming language. We made experiments using both SOT-DS candidates from Sect. 5.1 as underlying building blocks. We tested the performance of our implementations on an Intel Core i7 8th generation CPU, using the hash functions SHA-2, SHA-3, and HARAKA [9].[13] Table 1 shows the time consumption of the different algorithms.

**Table 1.** Efficiency of SDS (Fig. 9) based on two different SOT-DS (left: Fig. 7; right: Fig. 8) using three different hash functions. The entries indicate the number of $\mu$s per algorithm invocation, for a setting with $T = 100$ epochs.

| | **SHA-2** | **SHA-3** | **HARAKA** |
|---|---|---|---|
| *gen* | 67834 | 78227 | 80166 |
| *sig* | 528 | 557 | 545 |
| *ver* | 82 | 117 | 120 |
| *ext* | 162 | 276 | 239 |

| | **SHA-2** | **SHA-3** | **HARAKA** |
|---|---|---|---|
| *gen* | 75341 | 84401 | 88193 |
| *sig* | 643 | 699 | 664 |
| *ver* | 147 | 186 | 124 |
| *ext* | 283 | 373 | 239 |

When comparing the results associated with the two underlying SOT-DS schemes, the extra costs caused by the additional hash function operations of the second SOT-DS construction are clearly visible. Beyond that, we see that using SHA-2 is more efficient than using SHA-3. As HARAKA is specifically designed for handling fixed-length short inputs, it should be well suited for our application. However, HARAKA is also designed for modern platforms that support the AES-NI instruction set for AES [9]. Our implementation is generic and doesn't make use of such instructions, which makes it slower than our SHA-based candidates. We expect, however, that HARAKA will clearly outperform SHA-2 and SHA-3 with specifically optimized implementations.

### Acknowledgments

We thank the anonymous reviewers for their valuable comments.

[13] We borrowed the hash function code from the NIST Post-Quantum competition repository, see https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/example-files/api-notes.pdf.

## References


1. Bellare, M., Poettering, B., Stebila, D.: Deterring certificate subversion: Efficient double-authentication-preventing signatures. In: Fehr, S. (ed.) PKC 2017, Part II. LNCS, vol. 10175, pp. 121–151. Springer, Heidelberg (Mar 2017). https://doi.org/10.1007/978-3-662-54388-7_5
2. Bernstein, D.J., Hülsing, A., Kölbl, S., Niederhagen, R., Rijneveld, J., Schwabe, P.: The SPHINCS$^+$ signature framework. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 2129–2146. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3363229
3. Boneh, D., Kim, S., Nikolaenko, V.: Lattice-based DAPS and generalizations: Self-enforcement in signature schemes. In: Gollmann, D., Miyaji, A., Kikuchi, H. (eds.) ACNS 17. LNCS, vol. 10355, pp. 457–477. Springer, Heidelberg (Jul 2017). https://doi.org/10.1007/978-3-319-61204-1_23
4. BSI: Quantum-safe cryptography – Fundamentals, current developments and recommendations. Tech. rep., Bundesamt für Sicherheit in der Informationstechnik (2022), https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Brochure/quantum-safe-cryptography.html
5. Buchmann, J., Dahmen, E., Szydlo, M.: Hash-based digital signature schemes. In: Post-Quantum Cryptography. pp. 35–93. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-540-88702-7_3
6. Chaum, D., Fiat, A., Naor, M.: Untraceable electronic cash. In: Goldwasser, S. (ed.) CRYPTO'88. LNCS, vol. 403, pp. 319–327. Springer, Heidelberg (Aug 1990). https://doi.org/10.1007/0-387-34799-2_25
7. Derler, D., Ramacher, S., Slamanig, D.: Homomorphic proxy re-authenticators and applications to verifiable multi-user data aggregation. Cryptology ePrint Archive, Report 2017/086 (2017), https://eprint.iacr.org/2017/086
8. Derler, D., Ramacher, S., Slamanig, D.: Short double- and $N$-times-authentication-preventing signatures from ECDSA and more. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018. pp. 273–287. IEEE (2018). https://doi.org/10.1109/EuroSP.2018.00027
9. Kölbl, S., Lauridsen, M.M., Mendel, F., Rechberger, C.: Haraka v2 - Efficient short-input hashing for post-quantum applications. IACR Trans. Symm. Cryptol. **2016**(2), 1–29 (2016). https://doi.org/10.13154/tosc.v2016.i2.1-29, https://tosc.iacr.org/index.php/ToSC/article/view/563
10. Lamport, L.: Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory (Oct 1979)
11. Poettering, B.: Shorter double-authentication preventing signatures for small address spaces. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 18. LNCS, vol. 10831, pp. 344–361. Springer, Heidelberg (May 2018). https://doi.org/10.1007/978-3-319-89339-6_19
12. Poettering, B., Stebila, D.: Double-authentication-preventing signatures. In: Kutylowski, M., Vaidya, J. (eds.) ESORICS 2014, Part I. LNCS, vol. 8712, pp. 436–453. Springer, Heidelberg (Sep 2014). https://doi.org/10.1007/978-3-319-11203-9_25
13. Poettering, B., Stebila, D.: Double-authentication-preventing signatures. Int. J. Inf. Sec. **16**(1), 1–22 (2017). https://doi.org/10.1007/s10207-015-0307-8
14. Ruffing, T., Kate, A., Schröder, D.: Liar, liar, coins on fire!: Penalizing equivocation by loss of bitcoins. In: Ray, I., Li, N., Kruegel, C. (eds.) ACM CCS 2015. pp. 219–230. ACM Press (Oct 2015). https://doi.org/10.1145/2810103.2813686