# Virtual Smart Cards: How to Sign with a Password and a Server

Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin

IBM Research – Zurich
{jca, anj, nev, ksa}@zurich.ibm.com

**Abstract.** An important shortcoming of client-side cryptography on consumer devices is the poor protection of secret keys. Encrypting the keys under a human-memorizable password hardly offers any protection when the device is stolen. Trusted hardware tokens such as smart cards can provide strong protection of keys but are cumbersome to use. We consider the case where secret keys are used for digital signatures and propose a password-authenticated server-aided signature Pass2Sign protocol, where signatures are collaboratively generated by a device and a server, while the user authenticates to the server with a (low-entropy) password. Neither the server nor the device store enough information to create a signature by itself or to perform an offline attack on the password. The signed message remains hidden from the server. We argue that our protocol offers comparable security to trusted hardware, but without its inconveniences. We prove it secure in the universal composability (UC) framework in a very strong adaptive corruption model where, unlike standard UC, the adversary does not obtain past inputs and outputs upon corrupting a party. This is crucial to hide previously entered passwords and messages from the adversary when the device gets corrupted. The protocol itself is surprisingly simple: it is round-optimal, efficient, and relies exclusively on standard primitives such as hash functions and RSA. The security proof involves a novel random-oracle programming technique that may be of independent interest.

## 1 Introduction

Mobile devices such as smart phones and tablets are used more and more for security-critical tasks such as e-banking, authentication, and signing documents. However, they can be infected by malware and, due to their mobility, the devices are easily lost or stolen. Keeping cryptographic keys safe in such an environment is challenging. Typically, they are simply encrypted with a human-memorizable password. If a device is lost, stolen, or compromised by malware, the password-encrypted keys are usually easily recovered through an offline dictionary attack. Such attacks are extremely effective on modern hardware, especially given the low entropy in human-memorizable passwords [25].

Higher-security use cases such as online banking or government-issued electronic identification (eID) therefore often resort to tamper-proof hardware such

as smart cards, SIM cards or trusted platform modules (TPMs) for extra protection. The hardware tokens offer interfaces to interact with the keys, e.g., to compute digital signatures on messages provided by the host, while the signing key never leaves the confined environment. Usually, a password or PIN code is added as a second layer of protection. Offline attacks on the password or PIN are infeasible as the token blocks after too many failed attempts. In case a hardware token is compromised, it can additionally be rendered useless by revoking its public key.

The protection is not perfect though; without a dedicated display, malware on the host machine may instruct the plugged-in token to sign more or different messages than the user intended to sign. Also, side-channel attacks such as differential power analysis only become more powerful with time. In other words, what is considered tamper-proof hardware today, may not be so anymore tomorrow [31]. Additionally, trusted hardware suffers from poor usability and high deployment and maintenance costs. Users find it inconvenient to carry a hardware token for each security-sensitive application. Desktop and laptop computers rarely come with built-in smart card readers and not all consumer-grade machines have TPMs. External USB card readers are available, but supporting drivers and browser plug-ins on several platforms simultaneously requires a considerable effort. Using trusted hardware in combination with mobile devices is even more problematic, as they often lack connectivity to interact with external tokens.

So the question is, can we somehow realize similar security guarantees as hardware tokens, but avoid their practical inconveniences? Software obfuscation may come to mind, but does not help at all: leaking an obfuscated signing algorithm to an adversary is just as bad as leaking the signing key itself. As network connectivity is far more ubiquitous than trusted hardware in consumer devices, how about relying on the assistance of an online server to create signatures? A solution must protect the keys as long as at least one of the device or the server is not corrupted. Moreover, we want the user to remember at most a potentially weak password or a PIN code, while offering protection against offline password guessing attacks. Involving an online server in the signing process enables additional control of the use of the signing key, as the server can block the account or involve a second authentication factor when too many signing requests are made.

*Our primitive.* We introduce the notion of *password-authenticated server-aided signatures* (Pass2Sign), where the signing key is distributed over the user's device and an online server. The signing key is never reconstructed; rather, the device and the server must engage in a distributed protocol to compute signatures. For added security, the user must enter a password on the device each time a signature is generated. The server not only verifies the password, but also the identity of the device, i.e., an adversary without access to the device cannot even perform an online guessing attack. This prevents an adversary from blocking an honest user's account by swamping the server with fake login attempts: the server simply ignores signing attempts from the wrong device. If the device falls into

the wrong hands, or if the device is compromised by malware, then the attacker must still perform an *online* guessing attack before it can generate signatures. When the server detects too many failed password attempts or signing requests per time period, it can take appropriate action such as blocking the account or requiring additional authentication. The server neither learns the message that is being signed, nor does it learn the user's password (or password attempts). This not only protects the user against malicious servers, but also protects the password in case the server is broken into by hackers.

Malware running on the device can of course capture both the device keys and the password, enabling the adversary to sign any messages it wants, but only by interacting with the server for each new signature. The server can therefore implement additional security measures on top of our protocol, e.g., a logic which detects abnormal signing behavior, or a secondary communication channel via which the server informs the user about his account activity. When suspicious transactions are detected, the server can block the user's account to ensure that no further signatures can be created, and in case the activity indeed turns out to be malicious, revoke the user's public key.

The resulting security level is almost identical to the protection offered by trusted hardware tokens, but without their inconveniences. Only few smart card readers feature integrated trusted keypads and displays; built-in secure elements such as SIM cards or TPMs never do. Malware running on the host system can therefore also capture the user's PIN code and have different messages signed than what is shown on the screen. The main security guarantee of trusted hardware tokens is therefore that no more signatures can be generated after unplugging the token—which can actually be quite cumbersome or even impossible for SIM cards and TPMs. In the same way, our protocol prevents further signatures from being generated when the user's account is blocked by the server. When the device is lost, our solution even offers better protection than hardware: while it may be possible to extract the keys from a compromised token, the only information that an adversary can extract from a corrupted device is a useless key share.

*Strong security notion and corruption model.* We define the security of a Pass2Sign scheme in the universal composability (UC) framework [15]. The main goal of our protocol is to guarantee protection of the user's password and signing key in the event of device or server compromise. We therefore propose a very strong corruption model that, unlike standard corruptions as defined in the UC framework, does not hand all past inputs and outputs to the adversary when a party is corrupted. In case the device gets corrupted, these inputs include the user's password and all previously signed messages, which obviously goes directly against our security goals. Clearly, it is impossible to achieve such a strong corruption model without secure erasures: if the device cannot even erase the entered password, then there's no way to hide it from an adversary when the device is corrupted later.

The UC framework is well known to provide superior and more natural security guarantees for the particular case of password-based protocols than tradi-

tional game-based notions [14]. In particular, by letting the environment generate all passwords and password attempts, UC formulations correctly model arbitrary dependencies between passwords. For example, their game-based counterparts fail to provide any security guarantees when honest users make typos while entering their passwords, a rather frequent occurrence in real life. Also, by absorbing password guessing attacks inside the functionality, secure composition with other protocols is guaranteed to hold; this is much less clear for game-based notions that tolerate a non-negligible adversarial success probability.

*Efficient protocols.* One might expect that meeting such stringent security standards comes at a considerable cost in efficiency. Indeed, similar protocols involve a factor 4–10 in performance penalty to protect against (standard) adaptive corruptions [11], while generic techniques to obtain adaptive security at least double the number of communication rounds [37]. Blindness for signed messages is another feature that is notoriously expensive to achieve in the UC framework [30]. It is therefore even more surprising that our protocol is refreshingly simple, round-optimal, and efficient. Generating a signature requires only three modular exponentiations on the device and two on the server, plus a few hash function evaluations, with only one protocol message from the device to the server and back. We also describe two simpler variants of our protocol for the setting where message blindness is not required. Our first variant exposes the message only to the server, a second variant does not hide the message at all.

*New proof technique.* In spite of its simplicity, the security proof of the protocol is actually quite intricate. The many cases triggered by adaptive corruptions (which are allowed even during setup and arbitrarily interleaved signing sessions) and the mixture of passwords, encryption, and signatures require very careful bookkeeping, especially of the random-oracle responses during simulation. We employ an interesting and, to the best of our knowledge, novel technique to reconcile the seemingly contradictive requirements that the simulator must be able to determine the value of each signature, but without learning the message being signed. We avoid typical blind signature techniques and the associated "one-more"-type security assumptions [5] by letting the device and the server both contribute to the randomness of the signature, and by programming the random oracle "just-in-time" at the moment that the signature is verified, not when it is created. This technique is of independent interest, and may find applications in other scenarios as well.

*Implementation.* We implemented our protocol on a commodity mobile device and provide a thorough performance analysis of our prototypical implementation to demonstrate the practicality of our protocol. With signature generation for a 2048-bit key requiring roughly 250ms (including network delays), our protocol is clearly efficient enough for use in practice.

## 2 Related Work

The idea of distributing signing keys over two separate entities dates back to Boyd [10], and was later generalized to threshold signatures [18, 36, 22, 9, 2]. Threshold signatures assume that all parties somehow agree on the messages to be signed, i.e., there is no "main entity" that can trigger a signing protocol and thus also no authentication to ensure that only the "main entity" can do so. Bellare and Sandhu [7] present two-party RSA signature protocols between a client and a server. However, in their protocols the server cannot be corrupted and requires no client-authentication.

Server-assisted signatures with additional password protection have been presented in the literature as well. Ganesan [21] proposes a protocol where the client's signing exponent is derived from his password. This is similar to Gjøsteen and Thuen [24, 23] who propose password-based signature schemes where the secret signing key is shared between different entities, but where one share only depends on the password. A server knowing the high-entropy part of the signing key can mount offline attacks by creating a signature based on a guessed password (from which the low-entropy key share is derived) and verifying it under the public key. Hence, the above approaches assume that the server cannot be corrupted. He et al. [28] propose a related idea where a password is used to authenticate to a server, which, in turn, signs the message for the user. The scheme offers much weaker security guarantees than ours: the server can mount offline attacks against the user's password and sign messages in the name of the user, as it knows the signing key. Both is not possible in our scheme. Xu and Sandhu [38] present two server-assisted threshold signature schemes for a closely related setting where a user can generate the signatures with the help of his device and a threshold of multiple servers. Their constructions do not yield single-server instantiations as a special case, however, because in their setting a collusion of servers larger than the threshold can perform an offline attack on the user's password, without access to the user's device.

Another line of work are the schemes by Mannan and van Oorschot [33], and Damgård and Mikkelsen [17], which assume a trusted, yet resource-contrained device and aim at securely outsourcing parts of the device's computation to an untrusted entity. However, our goal is not to reduce the computational complexity for the trusted device, but to fully remove the assumption of trusted hardware.

The S-RSA protocol due to MacKenzie and Reiter [32] envisages many of the goals that we also pursue, such as requiring the adversary to compromise the device to perform even an online attack, avoiding offline attacks as long as the server and the device are not both compromised, and "key disabling" by blocking a user's account on the server. Their protocol is proven secure in a property-based (i.e., non-UC) notion that is weaker than ours in several respects. Foremost, it does not enjoy the many advantages of UC password-based protocols discussed earlier, such as preserving security in case of mistyped passwords and secure composition with other protocols. Also, the server in their protocol sees the message being signed, can only be corrupted between (and not during) signing

sessions, and can actually perform an offline dictionary attack on the password based on the information it sees during the signing protocol (but the last problem is easy to fix).

Our protocol is the first to support fully adaptive corruptions of the server as well as the device in the UC model. One could of course evaluate the signing algorithm using generic adaptively UC-secure multiparty computation (MPC), but this comes at great cost: evaluating even a *single* multiplication gate in the most efficient two-party computation protocol secure against adaptive corruptions [12] incurs computation and communication costs that are orders of magnitude larger than our direct construction.

A more remotely related primitive is password-authenticated secret sharing (PASS) [3]. A PASS-scheme allows a user to share a secret among a set of servers that it can later recover based on a password. In a sense, one could consider storing a secret signing key using a two-out-of-two PASS scheme, where the user's device plays the role of one of the servers. This does not fulfill our requirements, though, as the device would need to locally reconstruct the signing key in order to compute a signature. Moreover, once the signing key is reconstructed, the device can as many messages as it wants to without any interaction.

## 3 Preliminaries

Here we introduce the building blocks for our construction. These include RSA-FDH signatures ($\mathsf{DSIG}_{\mathsf{RSA}}$), non-committing encryption (NCE), trapdoor one-way permutations (TDP), and three UC-functionalities: $\mathcal{F}_{\mathsf{CA}}$ providing a PKI, $\mathcal{F}_{\mathsf{RO}}^{\mathcal{D}\to\mathcal{R}}$ modeling random-oracles, and $\mathcal{F}_{\mathsf{Auth}}$ providing authenticated channels.

*Notation.* We use $\tau \in \mathbb{N}$ as our security parameter. $1^\tau$ is the string of $\tau$ ones. All algorithms receive $1^\tau$ as an implicit input. $a \xleftarrow{r} S$ denotes that $a$ is assigned a random element chosen uniformly from the set $S$. If $A$ is a PPT algorithm we write $y \xleftarrow{r} A(x;r)$ to denote that $y$ is assigned the output of $A$ with input $x$ and external random coins $r$. If we drop $r$, the random coins are drawn internally. For deterministic algorithms, we write $y \leftarrow A(x)$. A function $\epsilon : \mathbb{N} \to \mathbb{R}$ is negligible if $\epsilon(\tau) = \tau^{-\omega(1)}$. By $|m|$ we denote the binary length of a message $m$. $|S|$ denotes the cardinality of the set $S$. If an argument is a list, we assume that the list has an injective encoding which allows to decode each element again.

*Ideal Functionality $\mathcal{F}_{\mathsf{CA}}$.* We assume a public-key infrastructure where servers can register their public keys, modeled by the ideal functionality $\mathcal{F}_{\mathsf{CA}}$ [13]. These keys can be retrieved by any party using the entity's identity for which the public key is requested. A formal definition of $\mathcal{F}_{\mathsf{CA}}$ is given in Figure 7 in the Appendix.

*Ideal Functionality $\mathcal{F}_{\mathsf{RO}}^{\mathcal{D}\to\mathcal{R}}$.* A random oracle can be seen as an idealized hash function that consistently maps inputs from domain $\mathcal{D}$ to random values in range $\mathcal{R}$ [6]. It is adjusted for our notation, as we parametrize it with domain $\mathcal{D}$ and range $\mathcal{R}$. We sometimes use more than one random oracle with the same range $\mathcal{R}$ and domain $\mathcal{D}$ for easier analysis, i.e., more than one random oracle corresponds

to $\mathcal{F}_{\mathsf{RO}}^{\mathcal{D}\to\mathcal{R}}$. To distinguish different random oracles, we assume that each call is prefixed with a unique identifier. The formal definition of $\mathcal{F}_{\mathsf{RO}}^{\mathcal{D}\to\mathcal{R}}$ is based on [29] and given in Figure 6 in the Appendix.

*Ideal Functionality $\mathcal{F}_{\mathsf{Auth}}$.* The $\mathcal{F}_{\mathsf{Auth}}$ functionality provides authenticated (but public) channels between parties [15, 13]. For our protocol, we can use the simplified version of [15], which allows to send a single authenticated message to the designated receiver. The formal definition of $\mathcal{F}_{\mathsf{Auth}}$ is given in Figure 8 in the Appendix.

*RSA Problem and RSA Assumption.* Let $(N, e, d, p, q) \xleftarrow{r} \mathsf{RSAGen}(1^\tau)$ be an RSA-key generator returning an RSA modulus $N = pq$, where $p$ and $q$ are random distinct primes, $e > 1$ an integer coprime to $\varphi(N)$, and $d \equiv e^{-1} \bmod \varphi(N)$. The RSA Problem associated to $\mathsf{RSAGen}$ is, given $N$, $e$, and $y \xleftarrow{r} \mathbb{Z}_N^*$, to find $x$ such that $x^e \equiv y \bmod N$. The RSA Assumption now states that for every PPT adversary $\mathcal{A}$, $\Pr[(N, e, d, p, q) \xleftarrow{r} \mathsf{RSAGen}(1^\tau), y \xleftarrow{r} \mathbb{Z}_N^*, x \xleftarrow{r} \mathcal{A}(N, e, y) : x^e \equiv y \bmod N] \leq \epsilon(\tau)$ for some negligible function $\epsilon$.

*RSA-FDH Signatures.* We use a RSA Full-Domain Hash (FDH) signature scheme $\mathsf{DSIG}_{\mathsf{RSA}} = (\mathsf{SKGen}_{\mathsf{RSA}}, \mathsf{Sign}_{\mathsf{RSA}}, \mathsf{Verify}_{\mathsf{RSA}})$ associated to RSA-key generator $\mathsf{RSAGen}$ defined as follows. The key generation algorithm $(spk, ssk) \xleftarrow{r} \mathsf{SKGen}_{\mathsf{RSA}}(1^\tau)$ runs $(N, e, d, p, q) \xleftarrow{r} \mathsf{RSAGen}(1^\tau)$ and outputs $ssk = (d, p, q)$ and $spk = (N, e)$. It also requires a hash function $\mathcal{H}_{\mathsf{RSA}} : \{0,1\}^* \to \mathbb{Z}_N^*$, modeled as a random oracle. To sign a message $m \in \{0,1\}^*$ with key $ssk = (d, p, q)$, $\mathsf{Sign}_{\mathsf{RSA}}$ computes $\sigma \leftarrow (\mathcal{H}_{\mathsf{RSA}}(m))^d \bmod N$. To verify if a signature $\sigma$ is valid for a message $m \in \{0,1\}^*$ and $spk = (N, e)$, $\mathsf{Verify}_{\mathsf{RSA}}$ outputs $\mathsf{true}$ if $0 < \sigma < N$ and $\mathcal{H}_{\mathsf{RSA}}(m) = \sigma^e \bmod N$, else it outputs $\mathsf{false}$. RSA-FDH signatures are strongly unforgeable against chosen message attacks in the random-oracle model if the RSA assumption holds [16].

*Trapdoor One-Way Permutations.* Let $(f, f^{-1}, \Sigma) \xleftarrow{r} \mathsf{TFGen}_{\mathsf{f}}(1^\tau)$ be the instance generator for a function $f : \Sigma \to \Sigma$ defining a permutation over $\Sigma$, with an inversion function $f^{-1} : \Sigma \to \Sigma$, such that we have 1) for all $x \in \Sigma$, all $\tau \in \mathbb{N}$, and for all $(f, f^{-1}, \Sigma) \xleftarrow{r} \mathsf{TFGen}_{\mathsf{f}}(1^\tau)$, we have $x = f^{-1}(f(x))$, and 2) for all PPT adversaries $\mathcal{A}$ we have $\Pr[(f, f^{-1}, \Sigma) \leftarrow \mathsf{TFGen}_{\mathsf{f}}(1^\tau), x \xleftarrow{r} \Sigma : x = \mathcal{A}(f, f(x), \Sigma)] \leq \epsilon(\tau)$ for some negligible function $\epsilon$. We also require that we can efficiently sample from $\Sigma$. An RSA-key generator $\mathsf{RSAGen}$ yields a trapdoor one-way permutation under the RSA assumption with $\Sigma = \mathbb{Z}_N^*$, $f(x) = x^e \bmod N$ and $f^{-1}(y) = y^d \bmod N$ [6].

*Non-Committing Labeled Public-Key Encryption Scheme.* To deal with adaptive corruptions, we require a non-committing encryption scheme. In the security proof, the simulator needs to be able to simulate ciphertexts without knowing the corresponding plaintexts which would be encrypted in the real protocol. However, when the adversary later corrupts the receiver of a simulated ciphertext, the simulator has to provide a state of the corrupted party such that all

**Experiment** $\mathsf{Exp}^{\mathsf{IND-NC-ideal}}_{\mathsf{NCE},\mathcal{A},\mathsf{SIM}_{\mathsf{NCE}}}(\tau)$:
$epk \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{publickey}, 1^\tau)$
$\mathcal{Q} \leftarrow \emptyset$
$state_{\mathcal{A}} \xleftarrow{r} \mathcal{A}^{\mathcal{O}^{\mathcal{H}(\cdot)},\mathcal{O}^{\mathsf{Enc}(\cdot,\cdot)},\mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}}(epk)$
where $\mathcal{O}^{\mathsf{Enc}(\cdot,\cdot)}$ on input $(m_i, \ell_i)$:
$C_i \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{encrypt}, |m_i|, \ell_i)$
$\mathcal{Q} \leftarrow \mathcal{Q} \cup \{(C_i, m_i, \ell_i)\}$
return $C_i$
where $\mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}$ on input $(C_j, \ell_j)$:
if $(C_j, m_j, \ell_j) \in \mathcal{Q}$, return $m_j$
else, return $m_j \leftarrow \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{decrypt}, C_j, \ell_j)$
where $\mathcal{O}^{\mathcal{H}(\cdot)}$ on input $q_k$:
return $h_k \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{roquery}, q_k)$
$esk \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{keyleak}, \mathcal{Q})$
return $\mathcal{A}^{\mathcal{O}^{\mathcal{H}(\cdot)}}(esk, state_{\mathcal{A}})$

**Experiment** $\mathsf{Exp}^{\mathsf{IND-NC-real}}_{\mathsf{NCE},\mathcal{A}}(\tau)$:
$(epk, esk) \xleftarrow{r} \mathsf{EKGen}(1^\tau)$
$state_{\mathcal{A}} \xleftarrow{r} \mathcal{A}^{\mathcal{O}^{\mathcal{H}(\cdot)},\mathcal{O}^{\mathsf{Enc}(\cdot,\cdot)},\mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}}(epk)$
where $\mathcal{O}^{\mathsf{Enc}(\cdot,\cdot)}$ on input $(m_i, \ell_i)$:
return $C_i \xleftarrow{r} \mathsf{Enc}(epk, m_i, \ell_i)$
where $\mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}$ on input $(C_j, \ell_j)$:
return $m_j \leftarrow \mathsf{Dec}(esk, C_j, \ell_j)$
where $\mathcal{O}^{\mathcal{H}(\cdot)}$ on input $q_k$:
return $h_k \xleftarrow{r} \mathcal{H}(q_k)$
return $\mathcal{A}^{\mathcal{O}^{\mathcal{H}(\cdot)}}(esk, state_{\mathcal{A}})$

**Fig. 1.** Experiments IND-NC-ideal and IND-NC-real for our IND-NC Definition.

the ciphertexts decrypt to some concrete plaintext. This is related to the "selective de-commitment problem" [4]. The notion of non-committing encryption that we require is stronger than some that were proposed in the literature [19, 27] and weaker than others [35]. To minimize the security assumptions for our protocol and leave open the possibility for more efficient instantiations, we introduce our own definition here and provide a non-interactive construction in the random-oracle model.

*A labeled non-committing encryption scheme* $\mathsf{NCE} = (\mathsf{EKGen}, \mathsf{Enc}, \mathsf{Dec})$ consists for three algorithms: a key generation algorithm $(epk, esk) \xleftarrow{r} \mathsf{EKGen}(1^\tau)$ outputting a public and secret key, where the public key specifies a finite message space $\mathcal{M}$, an encryption algorithm $C \xleftarrow{r} \mathsf{Enc}(epk, m, \ell)$ computing a ciphertext $C$ on input a public key $epk$, a message $m \in \mathcal{M}$, and a label $\ell \in \{0,1\}^*$, and a decryption algorithm $m' \leftarrow \mathsf{Dec}(esk, C, \ell)$ that takes as input a secret key $esk$, a ciphertext $C$ and a label $\ell$ and outputs either a message $m'$ or $\perp$ if decryption failed. We require the usual correctness properties to hold. Sometimes we need to explicitly talk about the random choices of the encryption algorithm. To this end, let $\Sigma$ be the space of these choices.

We now define the IND-NC security property that a labeled non-committing encryption scheme needs to satisfy in our context.

**Definition 1 (IND-NC security).** *An encryption scheme* $\mathsf{NCE} = (\mathsf{EKGen}, \mathsf{Enc}, \mathsf{Dec})$ *is IND-NC-secure iff for all PPT adversaries* $\mathcal{A}$ *there exists a stateful PPT simulator* $\mathsf{SIM}_{\mathsf{NCE}}$ *such that* $\left|\Pr[\mathsf{Exp}^{\mathsf{IND-NC-real}}_{\mathsf{NCE},\mathcal{A}}(\tau) = 1] - \Pr[\mathsf{Exp}^{\mathsf{IND-NC-ideal}}_{\mathsf{NCE},\mathcal{A},\mathsf{SIM}_{\mathsf{NCE}}}(\tau) = 1]\right| \leq \epsilon(\tau)$ *for some negligible function* $\epsilon$ *and the experiments of Figure 1.*

This definition says an encryption scheme is non-committing if there exists a simulator (that is given control over the random oracle) such that no adversary can distinguish between simulated ciphertexts (which do not contain any information about the plaintexts except their lengths) and real ones. More precisely,

the adversary cannot tell in which of the two experiments it is run, even if it can adaptively query for new encryptions and decryptions and receive the secret key at a later point. In both experiments, the adversary gets oracle access to the random oracle $\mathcal{H}$ and the encryption and decryption algorithms. At the end of both experiments, the adversary is handed the secret key used to encrypt all ciphertexts. The crucial difference is that in the ideal game the adversary gets handed simulated ciphertexts instead of real ones, which are computed by a simulator on input only the length of the plaintext. Only at the end of the game the simulator learns the plaintexts to which it produced the simulated ciphertexts. With that knowledge, the simulator then has to provide the adversary with the secret key such that the provided ciphertexts indeed decrypt to the messages that the adversary queried to the encryption algorithm.

Compared to the definition given by Fehr et al. [19], our adversary $\mathcal{A}$ is allowed adaptive queries and receives the secret key *esk* at the end of the experiment, while the definitions given in [27] only consider a single (randomly sampled) message per secret key, which is not enough for our protocol. Nielsen [35] gives a formulation of non-committing encryption—in the sense of secure message transmission—in the UC framework. Nielsen's functionality is stronger than what we need, however, because it also requires that the randomness used during encryption is simulatable, which we can avoid using secure erasures.

Analogously to [19], we show that our definition implies standard IND-CCA2 security. The proof of the following theorem is given in Appendix B, where we also recall a definition of IND-CCA2 security derived from [34].

**Theorem 1 (IND-NC $\Longrightarrow$ IND-CCA2).** *Every scheme which is IND-NC-secure, is also IND-CCA2-secure.*

*Instantiation.* We now give a concrete instantiation for an encryption scheme that achieves IND-NC security. We modify the CCA2-secure encryption scheme introduced in [6] to include labels and handle arbitrary-length messages. Let $\mathcal{G} : \{0,1\}^* \to \{0,1\}^\tau$ and $\mathcal{K} : \{0,1\}^* \to \{0,1\}^\tau$ denote two hash functions, modeled as random oracles. Let $\mathsf{ec} : \{0,1\}^* \to (\{0,1\}^\tau)^+$ be an injective encoding function and let $\mathsf{dc} : (\{0,1\}^\tau)^+ \to \{0,1\}^*$ denote the corresponding decoding function that returns $\perp$ if no valid pre-image exists. We require that the output length of $\mathsf{ec}$ only depends on the length of its input.

$\mathsf{EKGen}(1^\tau)$ : Generate a random trapdoor one-way permutation, i.e., $(f, f^{-1}, \Sigma) \xleftarrow{r} \mathsf{TFGen_f}(1^\tau)$. The message space $\mathcal{M}$ is $\{0,1\}^*$. Output the public key $epk = (f, \Sigma)$, and $esk = f^{-1}$ as the secret key.

$\mathsf{Enc}(epk, m, \ell)$ : Let $(m_1, m_2, \ldots, m_k) \leftarrow \mathsf{ec}(m)$. Draw $x \xleftarrow{r} \Sigma$, compute $C_1 \leftarrow f(x)$, $C_2^i \leftarrow \mathcal{G}(i, x) \oplus m_i$ for $i = 1, \ldots, k$, and $C_3 \leftarrow \mathcal{K}(x, k, m, \ell)$ and output the ciphertext $C := (C_1, (C_2^1, \ldots, C_2^k), C_3)$.

$\mathsf{Dec}(esk, C, \ell)$ : Parse $C$ as $(C_1, (C_2^1, \ldots, C_2^{k'}), C_3)$ for some $k' \geq 1$. Compute $x' \leftarrow f^{-1}(C_1)$ and $m_i' \leftarrow \mathcal{G}(i, x') \oplus C_2^i$ for $i = 1, \ldots, k'$. Let $m' \leftarrow \mathsf{dc}(m_1', \ldots, m'_{k'})$. If $m' = \perp$ or $C_3 \neq \mathcal{K}(x', k', m', \ell)$, output $\perp$, else output $m'$.

The above construction fulfills perfect correctness. We prove that the scheme described above achieves our notion of IND-NC-security in Appendix C.

**Theorem 2.** *The construction above is IND-NC-secure, if $\mathcal{G}$ and $\mathcal{K}$ are modeled as random oracles and $\mathsf{TFGen_f}(1^\tau)$ is a secure TDP generator.*

## 4 Ideal Functionality

We now formally define password-authenticated server-aided signatures (Pass2Sign) by describing its ideal functionality in the universal composability (UC) framework [15].

First, recall the high-level goal of our Pass2Sign scheme: signatures on messages should be derived collaboratively between two parties, a device $\mathcal{D}$ and a server $\mathcal{S}$, meaning that a valid signature can only be obtained if both parties agree to the generation. Access to the server's signing operation is protected by a user password *pwd* that is chosen at setup and needs to be provided for every signing request. The server then verifies whether the password is correct and also whether the request came from the correct device, which has the additional advantage that an attacker cannot block an honest user's account by swamping the server with false login attempts. The protocol must be secure against offline attacks on the password used during setup and on the password attempts during signing. That is, as long as at least one party remains honest, the adversary does not learn anything about the used passwords. In particular, the server learns only whether a password attempt in a signing request was correct or not, but not the actual password attempt itself. The server also does not learn the messages being signed. (If this blindness feature is not required, we discuss how it can easily be removed in Section 8.) Security must be guaranteed for adaptive corruptions in order to protect against the main threat, namely the user losing his device. Note that we subsume the user of the device into the environment to have a more readable functionality. How this maps to real-life scenarios is discussed at the end of this section.

The detailed description of our ideal functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$ for password-authenticated server-aided signatures is given in Figure 2. When describing our functionality, we use the following writing conventions to reduce repetitive notation:

- For the SETUPREQ and KEYGEN interfaces, the ideal functionality only considers the first input for each *sid*. Subsequent inputs to the same interface for the same *sid* are ignored. For the SIGNREQ, DELIVER, PROCEED, SIGNATURE interfaces the functionality only considers the first input for each combination of *sid* and *qid*.
- At each invocation, the functionality checks that $sid = (\mathcal{S}, \mathcal{D}, sid')$ for some server identity $\mathcal{S}$, device identifier $\mathcal{D}$ and $sid' \in \{0,1\}^*$. Also, whenever we say that $\mathcal{F}$ receives input from or provides output to $\mathcal{S}$ or $\mathcal{D}$, we mean $\mathcal{S}$ or $\mathcal{D}$ as specified in the *sid*, respectively.

1. `Setup Request Device.` On input $(\mathsf{SETUPREQ}, sid, pwd)$ from device $\mathcal{D}$:
   - Create a record $(\mathsf{setup\text{-}req}, sid, pwd)$ and send $(\mathsf{SETUPREQ}, sid)$ to $\mathcal{A}$.
2. `Key Generation.` On input $(\mathsf{KEYGEN}, sid, pwd^*, pk)$ from adversary $\mathcal{A}$:
   - Look up a record $(\mathsf{setup\text{-}req}, sid, pwd)$.
   - If $\mathcal{D}$ (taken from $sid$) is corrupt, then mark this instance as $\mathsf{key\text{-}corrupt}$.
   - If $\mathcal{D}$ is corrupt and $pwd^* \neq \perp$, then create a record $(\mathsf{setup}, sid, pwd^*, pk)$. Else, create a record $(\mathsf{setup}, sid, pwd, pk)$.
   - Output $(\mathsf{SETUP}, sid, pk)$ to $\mathcal{D}$.

---

3. `Sign Request.` On input $(\mathsf{SIGNREQ}, sid, qid, pwd', m)$ from device $\mathcal{D}$:
   - Look up a record $(\mathsf{setup}, sid, pwd, pk)$.
   - Create a record $(\mathsf{sign\text{-}req}, sid, qid, pwd', m)$.
   - Send $(\mathsf{SIGNREQ}, sid, qid)$ to $\mathcal{A}$.
4. `Sign Delivery.` On input $(\mathsf{DELIVER}, sid, qid, pwd^*, m^*)$ from adversary $\mathcal{A}$:
   - Look up records $(\mathsf{setup}, sid, pwd, pk)$ and $(\mathsf{sign\text{-}req}, sid, qid, pwd', m)$.
   - If $\mathcal{D}$ is corrupt and $pwd^* \neq \perp$, then set $pwd' \leftarrow pwd^*$ and $m \leftarrow m^*$.
   - If $pwd' = pwd$ then set $status \leftarrow \mathsf{pwdok}$, else set $status \leftarrow \mathsf{pwdwrong}$.
   - Create a record $(\mathsf{sign}, sid, qid, m, status)$.
   - Output $(\mathsf{SIGNREQ}, sid, qid, status)$ to $\mathcal{S}$.
5. `Server Proceed.` On input $(\mathsf{PROCEED}, sid, qid)$ from server $\mathcal{S}$:
   - Look up a record $(\mathsf{sign}, sid, qid, m, status)$ with $status = \mathsf{pwdok}$.
   - Update the record to $status \leftarrow \mathsf{proceed}$, and send $(\mathsf{PROCEED}, sid, qid)$ to $\mathcal{A}$.
6. `Signature Generation.` On input $(\mathsf{SIGNATURE}, sid, qid, \sigma)$ from $\mathcal{A}$:
   - Look up a record $(\mathsf{sign}, sid, qid, m, status)$.
   - If $\mathcal{S}$ is honest, only proceed if $status = \mathsf{proceed}$.
   - If there is no record $(\mathsf{signature}, pk, m, \sigma, \mathsf{false})$, then create a record $(\mathsf{signature}, pk, m, \sigma, \mathsf{true})$, and output $(\mathsf{SIGNATURE}, sid, qid, \sigma)$ to $\mathcal{D}$.

---

7. `Verify.` On input $(\mathsf{VERIFY}, sid, pk', m, \sigma)$ from a party $\mathcal{P}$:
   - Create a record $(\mathsf{verify}, sid, pk', m, \sigma)$ and send $(\mathsf{VERIFY}, sid, pk', m, \sigma, \mathcal{P})$ to $\mathcal{A}$.
8. `Verified.` On input $(\mathsf{VERIFIED}, sid, pk', m, \sigma, \phi)$ from $\mathcal{A}$ with $\phi \in \{\mathsf{true}, \mathsf{false}\}$:
   - Look up a record $(\mathsf{verify}, sid, pk', m, \sigma, \mathcal{P})$.
   - Record $(\mathsf{signature}, pk', m, \sigma, f)$ and output $(\mathsf{VERIFIED}, sid, pk', m, \sigma, f)$ to $\mathcal{P}$, where $f$ is determined as follows:
     - If a record $(\mathsf{signature}, pk', m, \sigma, f')$ for some $f'$ exists, set $f \leftarrow f'$. *(consistency)*
     - Else, if a record $(\mathsf{setup}, sid, pwd, pk)$ exists with $pk = pk'$ and the instance is not marked $\mathsf{key\text{-}corrupt}$, set $f \leftarrow \mathsf{false}$. *(strong unforgeability)*
     - Else, set $f \leftarrow \phi$.

**Fig. 2.** Main Interfaces of our Functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$.

- When we say that the functionality "looks up a record", we implicitly understand that if the record is not found, the functionality ignores the input and returns control to the environment.
- We assume that the session ($sid$) and query identifiers ($qid$) given as input to our functionality are globally unique. In the two-party setting that we consider, this can be achieved by exchanging random nonces between both parties and

9. `Corruption.` On input $(\mathsf{CORRUPT}, sid, \mathcal{P}, \Sigma)$ from adversary $\mathcal{A}$:
   - Look up a record $(\mathsf{setup}, sid, pwd, pk)$ and initialize a list $\mathcal{L} \leftarrow \emptyset$.
   - If $\mathcal{P} = \mathcal{S}$, then assemble $\mathcal{L}$ containing $(qid_i, c_i)$ for all existing records $(\mathsf{sign\text{-}req}, sid, qid_i, pwd'_i, m_i)$, where $c_i \leftarrow \mathsf{pwdok}$ if $pwd = pwd'_i$ and $c_i \leftarrow \mathsf{pwdwrong}$ otherwise.
   - If now both $\mathcal{D}$ and $\mathcal{S}$ are corrupt, then mark this instance as $\mathsf{key\text{-}corrupt}$ and complete the abandoned sign requests: For all $(qid_i, \sigma_i) \in \Sigma$, look up $m_i$ from record $(\mathsf{sign\text{-}req}, sid, qid_i, pwd'_i, m_i)$. If there does not exist a record $(\mathsf{signature}, pk, m_i, \sigma_i, \mathsf{false})$, then create a record $(\mathsf{signature}, pk, m_i, \sigma_i, \mathsf{true})$.
   - Send $(\mathsf{CORRUPT}, sid, \mathcal{P}, \mathcal{L})$ to $\mathcal{A}$.
10. `Password Guessing.` On input of $(\mathsf{PWDGUESS}, sid, qid, pwd^*)$ from adversary $\mathcal{A}$:
   - If not both $\mathcal{D}$ and $\mathcal{S}$ are corrupt, then ignore this input.
   - If $qid = \bot$ then look up a record $(\mathsf{setup\text{-}req}, sid, pwd)$.
   - If $qid \neq \bot$ then look up a record $(\mathsf{sign\text{-}req}, sid, qid, pwd, m)$.
   - Set $c \leftarrow \mathsf{pwdok}$, if $pwd^* = pwd$ and $c \leftarrow \mathsf{pwdwrong}$ otherwise.
   - Send $(\mathsf{PWDGUESS}, sid, qid, c)$ to $\mathcal{A}$.

**Fig. 3.** Corruption-Related Interfaces of our Functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$.

including the concatenation of both in the identifiers. We also assume that honest parties drop any inputs with session or query identifiers to which they did not contribute.

– When we say that an instance is "marked", we mean that the specified label is associated with the instance of the functionality with the current *sid*. This does not affect other instances of the functionality with a different *sid*.

We now describe the behavior of all interfaces in a somewhat informal manner to clarify the security properties that our functionality provides.

*Setup.* The setup-related interfaces allow the device $\mathcal{D}$ to create an account in $\mathcal{F}$ that is protected with a password *pwd* and associated with the server $\mathcal{S}$.

1. The $\mathsf{SETUPREQ}$ interface allows the device $\mathcal{D}$ to register with the server $\mathcal{S}$, where $\mathcal{D}$ and $\mathcal{S}$ are the identities as included in the session identifier.
2. The $\mathsf{KEYGEN}$ interface allows the adversary to complete the setup by determining the public key *pk* under which messages in this instance are signed. If the device $\mathcal{D}$ is corrupt at the time of key generation, then we say that the instance is $\mathsf{key\text{-}corrupt}$, meaning that the adversary may know the signing key and may therefore be able to sign any messages it wants. A corrupt device $\mathcal{D}$ additionally has the option to "overwrite" the original password *pwd* from the $\mathsf{SETUPREQ}$ input (that may have been provided when $\mathcal{D}$ was still honest) with a new password $pwd^*$. This does not affect the unforgeability of signatures, as the instance is $\mathsf{key\text{-}corrupt}$ anyway, but does allow the adversary to later perform signing requests with correct passwords with $\mathcal{S}$ without having to guess *pwd*.

*Signature Generation.* Once setup is completed, the signature-related interfaces allow the device $\mathcal{D}$ to obtain a signature $\sigma$ on a message $m$, but only if the provided password attempt $pwd'$ is correct and the server $\mathcal{S}$ agrees to the signature generation. Signing requests can be done in parallel; the unique query identifier $qid$ identifies different signing sessions.

3. The SIGNREQ interface allows the device $\mathcal{D}$ to submit the message $m$ to be signed, and a password attempt $pwd'$. Only the device $\mathcal{D}$ included in $sid$ can perform signing requests, so without compromising the device, the adversary cannot even perform an online attack against the setup password $pwd$.
4. The DELIVER interface lets the adversary notify the server of an incoming signing request. The notification only includes whether the submitted password is correct, but not the password attempt itself. A corrupt device $\mathcal{D}$ can overwrite the original password $pwd'$ and message $m$ from the SIGNREQ input (that may have been provided when $\mathcal{D}$ was still honest) with a new password $pwd^*$ and message $m^*$.
5. The PROCEED interface allows the server to indicate whether it wants to proceed with the signing request. This models the opportunity for an external throttling mechanism to refuse the signing request. An honest server can only proceed if the password was correct, but corrupt servers can proceed regardless of whether the passwords matched. If the server is honest, then the adversary only (implicitly) learns whether the password was correct when the server agrees to proceed.
6. The SIGNATURE interface allows the adversary to determine the value $\sigma$ of the signature and have it delivered to the requesting device. If the server is honest, then a signature can only be established if the server previously agreed to proceed. A corrupt server can choose to ignore an incorrect password. The functionality creates a signature record $(\mathsf{signature}, pk, m, \sigma, \mathsf{true})$ that will allow successful verification of the signature.

The above interfaces follow a similar approach to Canetti's signature functionality [13] where the adversary determines the signature value, with the important difference that the adversary does *not* learn the message being signed. This models a weak form of blindness: it ensures that a corrupt server does not learn the message, but it does not provide unlinkability as blind signatures do. A full blindness notion would let the functionality generate the signatures by running an algorithm provided by the adversary [20]. Achieving such a notion is interesting, but would almost certainly come at a considerable overhead, as is the case for standard (UC-secure) blind signatures [20, 30].

*Signature Verification.* With the verification interfaces, any party can check whether a signature $\sigma$ is valid for message $m$ and public key $pk'$.

7. The VERIFY interface allows any party $\mathcal{P}$ to ask for the verification of a signature $\sigma$ on message $m$ and under public key $pk'$.
8. The VERIFIED interface lets the adversary trigger the delivery of the verification result to $\mathcal{P}$ and also allows the adversary to input the verification result

$\phi$ for adversarially controlled keys $pk'$. Here, adversarially controlled means that either $pk'$ is different from the key $pk$ registered in the functionality, or the instance for $pk$ is key-corrupt. The ideal functionality enforces that responses are consistent, meaning that verification of the same signature for the same message and public key always returns the same result.

For a non-adversarially-controlled key $pk$, the functionality guarantees strong unforgeability, meaning that even if the message $m$ was signed before with signature $\sigma$, the adversary cannot come up with a different valid signature $\sigma' \neq \sigma$ for $m$. For regular unforgeability, one should add the condition that there does not exist a record $(\mathsf{signature}, pk', m, \sigma', \mathsf{true})$ for $\sigma' \neq \sigma$. We opt for strong unforgeability because it offers more application scenarios and implies regular unforgeability.

*Corruptions.* Our functionality supports adaptive corruptions, i.e., the environment can, at any time, decide to corrupt any initially honest party. It is *not* a standard-corruption functionality as defined by the UC framework [15] where the adversary, upon corruption of a party, obtains all the past inputs and outputs of that party. Such a corruption model is clearly unsuitable our setting, as it would hand the user password to the adversary as soon as the device gets corrupted. In fact, even when both the device and server are corrupted, we do not want to give away the passwords immediately. Instead, our functionality then offers an interface modeling offline attacks against the passwords.

9. The CORRUPT interface allows the adversary to dynamically corrupt an initially honest device or server. When the device is corrupted after setup was complete, the adversary obtains the possibility to perform signing requests, but he still needs to provide the correct password. We stress that the adversary doesn't receive any passwords attempts or messages from past or even ongoing signing protocols.

   When the server gets corrupted, the adversary is told for all past signing requests whether or not the respective password attempts were correct. Still, the adversary is not given the stored password, nor the past password attempts themselves. Also, the adversary is not able to generate valid signatures.

   When both the device and the server are corrupt, the instance becomes key-corrupt, meaning that the adversary can sign any messages that he wants, and the adversary can offline-attack past passwords using the PWDGUESS interface described below. Additionally, the adversary can finish "abandoned" sign requests, meaning sign requests that were never delivered to the server, that were overwritten by the adversary, that were turned down by the server because the password was incorrect, or for which the server did not (yet) agree to proceed. The adversary determines the signature values for abandoned requests in a separate input $\Sigma$. This interface may seem superfluous now that the instance is key-corrupt anyway, but the the adversary does not know the message of sign requests that were initiated when the device was still honest, so it cannot register these signatures through the normal VERIFIED interface. In the real world, an adversary who obtains the full state information of the

device and server can inherently complete abandoned queries, so we have to model it here as well.

10. The PWDGUESS interface allows the adversary to perform offline attacks on the stored password and on previous password attempts, but only becomes available when both the device and the server are corrupt. For the stored password, offline attacks cannot be avoided, as the device and the server together must store some information that allows them to decide whether a password attempt was correct. For previous password attempts, this could in principle be avoided, but would make our protocol considerably less efficient, because new cryptographic material would have to be generated at each request and securely deleted afterward.

If the device is already corrupt at the time of setup, we consider the instance as key-corrupt even though the server might still be honest. A stronger security notion, requiring only slight changes to the functionality, would be achievable where the instance is only considered key-corrupt when both the device and server are corrupted. However, this would mean that in the realization, the key generation be done distributively between the server and the device. This is possible but for RSA rather inefficient [1, 26] and seems to offer little added security; hence we chose not to do this.

*Discussion.* Let us discuss how real-world attack scenarios map to our ideal functionality. If a user loses his device, we assume that the adversary is able to extract *all* the data from the device, so the device becomes corrupted. As long as the server is not corrupted, though, the adversary controlling the device still has to make online password guesses to be able to sign, but does not obtain the (full) signing key. To protect against online password guessing, the server should implement some kind of throttling on top of our protocol, such as refusing to serve further queries after too many failed password attempts.

If the device becomes infected by malware, we also capture the worst case scenario: it may get *all* the data from the device and hence the device becomes corrupted. In contrast with the scenario above, the malware may also learn the (correct) password of the user if he's unaware of the infection and continues to use the device. This behavior is subsumed into the environment; we model this correctly by letting the environment provide the correct password to the adversary. Some protection against this kind of attack can be implemented on top of our protocol by adding intrusion detection logic on the server's side, e.g., by stopping to serve requests if they become too frequent. This situation is actually quite similar to that of a smart card inserted in an infected device: the device could intercept the PIN code and sign any messages it wants until the card is removed.

One could consider a more gradual corruption model where the device can be semi-corrupted, e.g., if an application turns malicious, but the uncompromised operating system separates it from other applications on the device. Our model covers this as long as applications have their own protected execution space: the device in our model represents the application, while everything else is subsumed

into the environment. More advanced models where applications can observe other applications (e.g., their running times) are beyond the scope of this paper.

## 5 Our Pass2Sign Protocol

The core idea of our protocol is fairly simple: an RSA secret key $d = d_{\mathcal{D}} \cdot d_{\mathcal{S}} \bmod \varphi(N)$ is split between the device and the server who then jointly perform the signing operation for each message $m$. To hide the message from the server, the device "blinds" it with randomness $r$ as $h_m \leftarrow \mathcal{H}(r, m)$ and lets the server sign it as $\sigma_{\mathcal{S}} \leftarrow h_m^{d_{\mathcal{S}}}$. The device completes the signature as $\sigma \leftarrow \sigma_{\mathcal{S}}^{d_{\mathcal{D}}}$. For each signing request, the user authenticates towards the server using a salted password hash $h_p \leftarrow \mathcal{H}(k, pwd)$, where the salt $k$ is stored on the device.

*Our corruption model and the need for secure erasures.* The main challenge is to maintain this simplicity while achieving the strong security properties that we envisage. Most often, security against adaptive corruptions in the UC model comes at a considerable price in terms of computation and communication, and our corruption model is even substantially stronger. In particular, recall that we want to protect the user's password and previously signed messages in case the device is lost or stolen. The "standard corruption" model in the UC framework [15] hands all previous inputs and outputs of a party to the adversary upon corruption of that party, which in case of the device would include all previous passwords and messages. It is quite obvious that standard corruption does not suffice for our purposes, and also that our model cannot be achieved without secure erasures, as there would be no way to securely erase previous inputs. Given the usual difficulty of achieving even standard UC corruption, it is surprising that our protocol remains refreshingly simple, round-optimal, and efficient.

*Achieving blindness.* Achieving blind signatures against adaptive corruptions in the UC model is notoriously hard: the only scheme is due to Kiayias and Zhou [30] and requires six rounds of communication and several zero-knowledge proofs. We decided to strike a reasonable compromise between security and efficiency by dropping the unlinkability requirement, i.e., the property that the signer cannot link a signature to a previous signing transcript, but focusing entirely on hiding the message from the signer. We describe a new "just-in-time" programming technique for the random oracle that inserts the correct entries into the oracle when signatures are verified, rather than when they are created. We thereby obtain an efficient and round-optimal construction without having to rely on one-more-type assumptions that are typical for full-domain-hash blind signatures [5, 8].

In a bit more detail, to enable the simulator to open any signing transcript to any message-signature pair, the server adds another layer of randomness, i.e., he signs $h'_m \leftarrow \mathcal{H}(r', h_m)$ for some randomly chosen $r'$. When the simulator has to provide a signature $\sigma$ to the functionality without knowing the message $m$, it simply signs a random value $h'_m \xleftarrow{r} \{0,1\}^\tau$. The connection to $m$ is only

established when the signature is verified, which we call "just-in-time" programming. Namely, whenever a random oracle query $\mathcal{H}(r', \mathcal{H}(r, m))$ is made where $r, r'$ were previously used in a simulated blind signature, the simulator verifies whether $(m, \sigma)$ is valid with the help of the ideal functionality. If so, the simulator programs the random oracle to map the message $m$ it just learned to the randomly chosen $h'_m$ that was signed as $\sigma$.

*Non-committing communication and state.* As we allow corruptions *during* setup and signing sessions, we have to take special care that messages sent by the device and server do not commit the simulator to values that it might not know at that time in the proof. We achieve this by employing non-committing encryption for the passwords hashes $h_p \leftarrow \mathcal{H}(k, pwd)$ and each password attempt $h'_p \leftarrow \mathcal{H}(qid, \mathcal{H}(k, pwd'))$ that the device sends to the server. At a first glance that might seem unnecessary since we also assume secure erasures. However, secure erasures are not sufficient as an adversary can intercept the ciphertexts and later corrupt the server to learn the decryption key. He then expects all ciphertexts to open to the proper password hashes (that in the security proof might be unknown when the ciphertexts are generated). The non-committing encryption gives us exactly that flexibility. To determine the correct password hashes $h_p$ and $h'_p$ upon server corruption we use different random oracle programming techniques, eventually also reying on the password guessing interfaces of the ideal functionality (if both parties are corrupted).

We have to take similar care for the intermediate state-records that the device keeps during interactive protocols. After sending a signing request, the device cannot store the message $m$, or even the randomness $r$ and the message hash $h_m \leftarrow \mathcal{H}(r, m)$, as the simulator does not learn $m$ upon corrupting the device. Nevertheless, the device must be able to verify whether the server's contribution is correct. Therefore, when sending the message hash $h_m$, the device also sends a value $t \leftarrow \mathcal{H}(\texttt{"MAC"}, qid, k, h_m)$ that acts as a message authentication code (MAC) for $h_m$. This allows the device to check that the server signed the correct message upon receiving the signature share, but without requiring state information that depends on $m$.

*Authentication of participants.* As already mentioned earlier, the session identifier *sid* contains the identities of the device $\mathcal{D}$ and the server $\mathcal{S}$. This means that $\mathcal{D}$ and $\mathcal{S}$ have to be authenticated. We do so by employing $\mathcal{F}_{\mathsf{Auth}}$ for authenticated communication, thereby making abstraction of how the authentication is performed. This could be through a shared secret, through digital signatures (e.g., TLS with client authentication), or in an "offline fashion" by letting the user use a trusted third party to register the device, such as a bank or a local municipal office. The last option has the additional advantage that one could also check the name or other credentials of the user, and also directly certify the resulting public key of the user.

### 5.1 Protocol Description

We now present the detailed protocol for our Pass2Sign scheme and give a simplified presentation in Figures 4 and 5. We assume that a server has a key pair $(epk, esk)$ for a non-committing encryption scheme $(\mathsf{EKGen}, \mathsf{Enc}, \mathsf{Dec})$, generated by $\mathsf{EKGen}$ on input the security parameter $1^\tau$. We also assume a public-key infrastructure, where devices and servers can register their public keys, modeled by the ideal functionality $\mathcal{F}_{\mathsf{CA}}$ [13], and authenticated message transmission, modeled by $\mathcal{F}_{\mathsf{Auth}}$. In the protocol description we denote inputs to and outputs from them informally to make the protocol more readable (e.g., we will write that $\mathcal{S}$ sends $m$ to $\mathcal{D}$ via $\mathcal{F}_{\mathsf{Auth}}$ instead of an explicit call to $\mathcal{F}_{\mathsf{Auth}}$ with sub-session IDs etc.). We further assume that parties check the correctness of session and sub-session IDs in all inputs. Moreover, we use $\mathcal{H}$ and $\mathcal{H}_{\mathsf{RSA}}$ as shorthand notations for two random-oracle functionalities $\mathcal{F}_{\mathsf{RO}}^{\{0,1\}^*\to\{0,1\}^\tau}$ and $\mathcal{F}_{\mathsf{RO}}^{\{0,1\}^*\to\mathbb{Z}_N^*}$, respectively. Note that these are single-instance functionalities; one can obtain a secure multi-instance implementation by prefixing each call to them with $sid$. Our protocol further makes use of an RSA-key generator $\mathsf{RSAGen}$.

As discussed earlier, secure erasures are necessary to achieve our security guarantees. We thus assume that after each protocol step all variables are deleted except unless we explicitly state that a variable is stored.

Finally, we assume that whenever a check performed by the server or device fails, the checking party will abort the protocol.

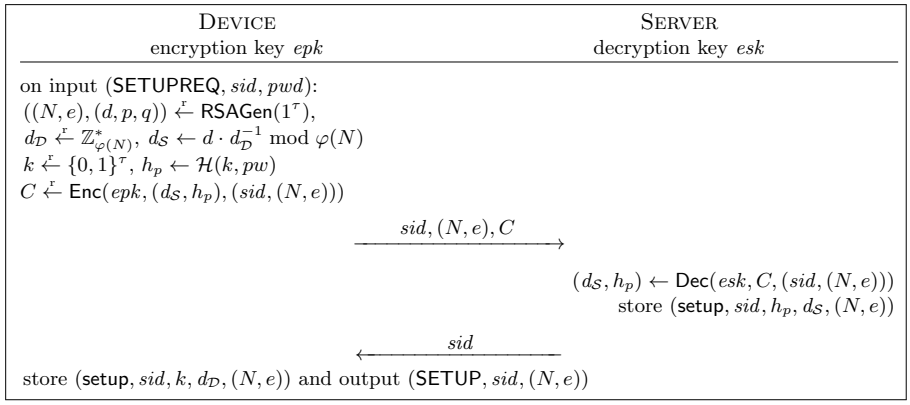


**Fig. 4.** Setup Protocol *(simplified)*.

*Setup Protocol.* The setup procedure is the following protocol that a device $\mathcal{D}$ runs on input $(\mathsf{SETUPREQ}, sid, pwd)$ with server $\mathcal{S}$, where $sid = (\mathcal{S}, \mathcal{D}, sid')$.

**Setup – Step 1. Device generates account data:**

a) Upon input $(\mathsf{SETUPREQ}, sid, pwd)$, retrieve the encryption key $epk$ for $\mathcal{S}$ from $\mathcal{F}_{\mathsf{CA}}$.

b) Generate RSA key material as $(N, e, d, p, q) \xleftarrow{r} \mathsf{RSAGen}(1^\tau)$ and share the secret exponent $d$ by choosing a random $d_\mathcal{D} \xleftarrow{r} \mathbb{Z}^*_{\varphi(N)}$ and setting $d_\mathcal{S} \leftarrow d \cdot d_\mathcal{D}^{-1} \bmod \varphi(N)$, where $d_\mathcal{S}$ is encoded as an $|N|$-bit string.
c) Compute a "salted" password hash as $h_p \leftarrow \mathcal{H}(k, pwd)$ for a random $k \xleftarrow{r} \{0,1\}^\tau$.
d) Encrypt the RSA key share $d_\mathcal{S}$ and the authentication information $h_p$ under $epk$ and with the label $(sid, (N, e))$. That is, compute $C \xleftarrow{r} \mathsf{Enc}(epk, (d_\mathcal{S}, h_p), (sid, (N, e)))$.
e) Store the record $(\mathsf{setup\text{-}temp}, sid, k, d_\mathcal{D}, (N, e))$ and send the message $m = (sid, (N, e), C)$ to the server $\mathcal{S}$ using $\mathcal{F}_\mathsf{Auth}$.

**Setup – Step 2. Server registers account:**

a) Upon receiving $m = (sid, (N, e), C)$ from $\mathcal{D}$ via $\mathcal{F}_\mathsf{Auth}$, check that $sid$ is not registered yet.
b) Decrypt $C$ as $(d_\mathcal{S}, h_p) \leftarrow \mathsf{Dec}(esk, C, (sid, (N, e)))$. If decryption succeeds, store $(\mathsf{setup}, sid, h_p, d_\mathcal{S}, (N, e))$.
c) Acknowledge the created account by sending $(sid)$ to $\mathcal{D}$ via $\mathcal{F}_\mathsf{Auth}$.

**Setup – Step 3. Device completes registration:**

a) Upon receiving a message $(sid)$ from $\mathcal{S}$ via $\mathcal{F}_\mathsf{Auth}$, check that a record $(\mathsf{setup\text{-}temp}, sid, k, d_\mathcal{D}, (N, e))$ for $sid$ exists.
b) Store $(\mathsf{setup}, sid, k, d_\mathcal{D}, (N, e))$ and end with output $(\mathsf{SETUP}, sid, (N, e))$.

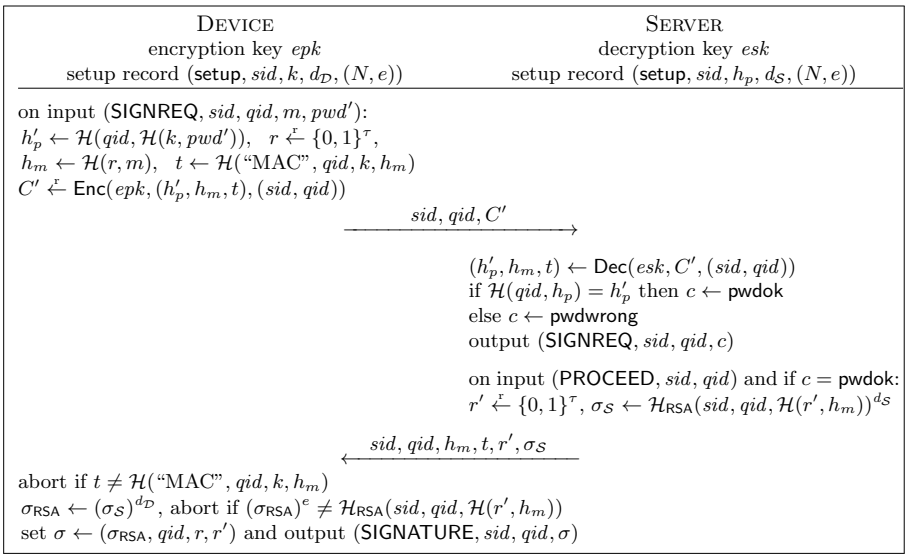


**Fig. 5.** Signing Protocol *(simplified)*.

*Signing Protocol.* The signing protocol starts when the device $\mathcal{D}$ receives an input $(\mathsf{SIGNREQ}, sid, qid, m, pwd')$, where $sid = (\mathcal{S}, \mathcal{D}, sid')$, upon which he runs the following protocol with the server $\mathcal{S}$. Recall that we assume that both parties have previously agreed upon a common and globally unique query identifier $qid$. All messages sent between the device and server also contain the $qid$ as prefix, and only those messages with the corresponding $qid$ are further processed.

**Sign – Step 1. Device sends signing request:**

a) Upon input $(\mathsf{SIGNREQ}, sid, qid, m, pwd')$, retrieve the record $(\mathsf{setup}, sid, k, d_{\mathcal{D}}, (N, e))$ for $sid$.
b) "Blind" the message by drawing $r \xleftarrow{r} \{0,1\}^{\tau}$ and computing $h_m \leftarrow \mathcal{H}(r, m)$.
c) Compute the (re-)authentication value $h'_p \leftarrow \mathcal{H}(qid, \mathcal{H}(k, pwd'))$.
d) Compute a "MAC" $t$ of $h_m$ as $t \leftarrow \mathcal{H}(\texttt{"MAC"}, qid, k, h_m)$.
e) Generate a non-committing encryption of $h'_p$, $h_m$, and $t$ under the public key $epk$ and with label $(sid, qid)$ as $C' \xleftarrow{r} \mathsf{Enc}(epk, (h'_p, h_m, t), (sid, qid))$.
f) Store the record $(\mathsf{sign}, sid, qid, r)$ and send $(sid, qid, C')$ to $\mathcal{S}$ via $\mathcal{F}_{\mathsf{Auth}}$.

**Sign – Step 2. Server verifies information:**

a) Upon receiving $(sid, qid, C')$ from $\mathcal{D}$ via $\mathcal{F}_{\mathsf{Auth}}$, retrieve the record $(\mathsf{setup}, sid, h_p, d_{\mathcal{S}}, (N, e))$ for $sid$.
b) Decrypt $C'$ to $(h'_p, h_m, t) \leftarrow \mathsf{Dec}(esk, C', (sid, qid))$.
c) Check the password by verifying whether $\mathcal{H}(qid, h_p) = h'_p$ and set $c \leftarrow \mathsf{pwdok}$ if so and $c \leftarrow \mathsf{pwdwrong}$ otherwise.
d) Store the record $(\mathsf{sign}, sid, qid, h_m, t, c)$ and output $(\mathsf{SIGNREQ}, sid, qid, c)$.

**Sign – Step 3. Server creates its signature share:**

a) Upon input $(\mathsf{PROCEED}, sid, qid)$, retrieve $(\mathsf{sign}, sid, qid, h_m, t, c)$ for $qid$ and abort if $c \neq \mathsf{pwdok}$.
b) Compute the signature share $\sigma_{\mathcal{S}} \leftarrow \mathcal{H}_{\mathsf{RSA}}(sid, qid, \mathcal{H}(r', h_m))^{d_{\mathcal{S}}} \bmod N$ for a random $r' \xleftarrow{r} \{0,1\}^{\tau}$.
c) Send $(sid, qid, h_m, t, r', \sigma_{\mathcal{S}})$ to $\mathcal{D}$ via $\mathcal{F}_{\mathsf{Auth}}$.

**Sign – Step 4. Device completes the signature:**

a) Upon receiving $(sid, qid, h_m, t, r', \sigma_{\mathcal{S}})$ from $\mathcal{S}$ via $\mathcal{F}_{\mathsf{Auth}}$, retrieve the signing record $(\mathsf{sign}, sid, qid, r)$ for $qid$ and setup record $(\mathsf{setup}, sid, k, d_{\mathcal{D}}, (N, e))$.
b) Verify that $t = \mathcal{H}(\texttt{"MAC"}, qid, k, h_m)$.
c) Complete the signature by computing $\sigma_{\mathsf{RSA}} \leftarrow (\sigma_{\mathcal{S}})^{d_{\mathcal{D}}} \bmod N$. Verify that $(\sigma_{\mathsf{RSA}})^e = \mathcal{H}_{\mathsf{RSA}}(sid, qid, \mathcal{H}(r', h_m)) \bmod N$ holds, i.e., that the server's signature share was correct.
d) Set $\sigma \leftarrow (\sigma_{\mathsf{RSA}}, qid, r, r')$ and end with output $(\mathsf{SIGNATURE}, sid, qid, \sigma)$.

*Signature Verification.* On input $(\mathsf{VERIFY}, sid, m, \sigma, pk)$, parse $pk = (N, e)$, $\sigma = (\sigma_{\mathsf{RSA}}, qid, r, r')$ and set $M \leftarrow (sid, qid, \mathcal{H}(r', \mathcal{H}(r, m)))$. If $\sigma_{\mathsf{RSA}}$ is a valid RSA signature on $M$, i.e., if $0 < \sigma_{\mathsf{RSA}} < N$ and $\mathcal{H}_{\mathsf{RSA}}(M) = \sigma_{\mathsf{RSA}}^e \bmod N$, output $(\mathsf{VERIFIED}, sid, m, \sigma, pk, \mathsf{true})$ and $(\mathsf{VERIFIED}, sid, m, \sigma, pk, \mathsf{false})$ otherwise.

## 6 Security

The full proof of the following theorem is given in Appendix D, we give a sketch below.

**Theorem 3.** *The* Pass2Sign *scheme described in Section 5 securely implements the ideal functionality* $\mathcal{F}_{\mathsf{Pass2Sign}}$ *defined in Section 4 in the* $(\mathcal{F}_{\mathsf{CA}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{Auth}})$*-hybrid model with secure erasures if the RSA assumption associated to* RSAGen *holds and* $(\mathsf{EKGen}, \mathsf{Enc}, \mathsf{Dec})$ *is an IND-NC secure encryption scheme.*

Using the IND-NC secure encryption scheme proposed in Section 3, which is an extension of the Bellare-Rogaway CCA2 encryption scheme, and instantiated with the RSA trapdoor permutation, we get the following corollary:

**Corollary 1.** *The* Pass2Sign *scheme described in Section 5 and instantiated as described above, securely implements the ideal functionality* $\mathcal{F}_{\mathsf{Pass2Sign}}$ *defined in Section 4 in the* $(\mathcal{F}_{\mathsf{CA}}, \mathcal{F}_{\mathsf{RO}}, \mathcal{F}_{\mathsf{Auth}})$*-hybrid model with secure erasures if the RSA assumption associated with* RSAGen *holds.*

*Proof (Sketch).* The proof of Theorem 3 is done by providing a simulator and a sequence of games. The initial game is the real experiment and the final game (Game 10) runs the simulator given only the information that is also available to the adversary when interacting with the ideal functionality. We now give a rough sketch of the game sequence, the detailed description as well as the description of the simulator is given in Appendix D.

Game 1 through and Game 2 abort when collisions occur in random-oracle outputs, or if the adversary "predicts" random-oracle outputs. Game 3 replaces all ciphertexts sent by an honest device to an honest server with simulated "dummy" ciphertexts. It uses the decryption simulation to decrypt ciphertexts that were not sent by the honest device, and uses the key-leakage simulation to obtain the secret key *esk* in case the server is corrupted.

In Game 4 and Game 5, the simulator aborts when a valid signature is verified that did not originate from the device or the server, whichever is still honest. Interestingly, the case for an honest device can be reduced from the unforgeability of RSA-FDH, but for an honest server we have to reduce straight from the RSA assumption (similar as in [7]).

The subsequent games are all about making the setup and signing protocol simulations independent of the actual values of the passwords and messages being signed. Game 6 and Game 7 make the setup and signing protocol simulations independent of the actual value of the password and password attempts. When the server is corrupt, instead of deriving $h_p$ and $t$ from the device secret $k$ and the real passwords, the simulator uses random values. For $h'_{p,i}$ it either uses $\mathcal{H}(qid, h_p)$ or a random value, depending whether the password attempt used in a signing request is correct or not. As long as the device is honest, this change cannot be detected by the adversary as he doesn't know $k$. When the device gets corrupted too, and thus the adversary learns $k$, the simulator starts programming the random oracle consistenly to the previously chosen $h_p$ and $h'_{p,i}$ whenever the

actual passwords $pwd$ and $pwd'_i$ are queried. In the final simulation with the ideal functionality this is done with the help of the password guessing interface that becomes available as soon as both, the device and the server, are corrupted.

If both entities are initially honest and the device gets corrupted first, the simulation is different though. Then we keep $h_p, h'_{p,i}$ unassigned and only fix their values when the server gets corrupted too. This is sufficient as the device never stores the hash values and sends them only in encrypted form to the server (which are dummy ciphertexts since Game 3). However, as soon as the server gets corrupted, the adversary learns the secret key of the encryption scheme and thus all previous dummy communication between the device (sent by the simulator when $\mathcal{D}$ was still honest) and the server must decrypt to the correct hash values. Here we cannot assign random values to $h_p, h'_{p,i}$ though. This stems from the fact that the device got corrupted first, and thus the adversary knows the device secret $k$ and could have already computed $h_p$ or $h'_{p,i}$ for the correct password values $pwd, pwd_i$. Thus, in order to ensure consistency, we check whether a previously answered random oracle query contained an actual password $pwd$ or $pwd'_i$ (when switching to the ideal world, this is done via the password guessing interface). If such a query is found, we simply reuse the previously given random oracle response for $h_p$ or $h'_{p,i}$ and assign random values otherwise. We then use the keyleak-simulator of the non-committing encryption scheme to obtain a key $esk$ that decrypts the dummy ciphertexts from Game 3 to the just determined hash values. In fact, this case is the reason why we need *non-committing* encryption, as in the ideal world the passwords $pwd$ or $pwd'_i$ are unknown to the simulator (even when the device gets corrupted), and the password guessing interface only becomes available when both entities got corrupted.

Game 8 and Game 9 make the signing process independent of the message $m$, with an interesting technique that programs the relevant random-oracle entries only when the signature is verified, not when it is created. In Game 8, an honest device interacting with a corrupt server chooses random values $r$ and $h_m$ and, if the server behaves honestly, computes the signature as $\sigma_{\mathsf{RSA}} = \mathcal{H}_{\mathsf{RSA}}(sid, qid, \mathcal{H}(r', h_m))^{d_{\mathcal{S}}\, d_{\mathcal{D}}} \bmod N$, which it can compute because it knows the keys $d_{\mathcal{S}}$ and $d_{\mathcal{D}}$ from the time they were generated. The simulator stores the resulting signature $\sigma = (\sigma_{\mathsf{RSA}}, qid, r, r')$ as valid for $m$ in its records; in the ideal world, the simulator would input $(\mathsf{SIGNATURE}, sid, qid, \sigma)$ to let the functionality associate $\sigma$ to $m$. When later a random-oracle query $\mathcal{H}(r, m)$ comes in, the simulator looks up whether a signature $\sigma$ was recorded with randomness $r$, and if so, checks whether $\sigma$ has been recorded as a valid signature for $m$ (using its own records, or using the $\mathsf{VERIFY}$ interface in the ideal world). If so, then it uses the value $h_m$ used during the corresponding signing protocol as the random-oracle response, otherwise it returns a random value. Game 9 acts similarly when the device and server are both honest, but by letting the server sign random $h'_m$ values, without knowing the corresponding $h_m$. Only when a query $\mathcal{H}(r', h_m)$ is made, $h'_m$ is assigned if a corresponding signature has been recorded.

## 7 Implementation of Our Pass2Sign Scheme

In this section we give a short summary of our prototypical implementation of the Pass2Sign scheme. A more detailed description is given in Appendix E. We measured our protocol with three different RSA-moduli sizes, 1,024, 2,048 and 4,096Bit to account for different security requirements. The key size is used for both the signing key and the trapdoor permutation in the non-committing encryption scheme. To instantiate the random oracles $\mathcal{K}, \mathcal{G}$, and $\mathcal{H}$ we use SHA-512 and prefix each call accordingly. The instantiation of the full-domain hash $\mathcal{H}_{\mathsf{RSA}}$ is based on the construction given in [6], and uses rejection sampling to uniformly map into $\mathbb{Z}_N^*$. The communication partners send messages using standard TCP-Sockets.

Our implementation uses Java 8 without any optimization. Our server is a laptop with a 2.7GHz processor and 16GB RAM, while the device is a Nexus 10 tablet with 1.7GHz, 2GB RAM and Android 5.1.1.

Table 1 depicts the average time for the setup and signing protocol, split between the device and server part, and based on measurements of 100 independent protocol runs. The figures do not include network latencies though, as they strongly depend on the actual location setting. However, assuming a round-trip time takes 100ms, a full signing protocol with 2,048Bit keys then requires roughly 250ms in total.

| | Setup | | | Signing | | |
|---|---|---|---|---|---|---|
| Key Size | 1,024 Bit | 2,048 Bit | 4,096 Bit | 1,024 Bit | 2,048 Bit | 4,096Bit |
| **Device** | | | | | | |
| Median | 648.11 | 3′335.34 | 14′343.46 | 19.08 | 79.83 | 482.60 |
| Average | 855.58 | 3′646.27 | 16′202.58 | 19.79 | 83.40 | 574.41 |
| **Server** | | | | | | |
| Median | 14.32 | 63.96 | 388.11 | 11.76 | 64.53 | 456.38 |
| Average | 15.20 | 65.69 | 393.27 | 12.31 | 65.50 | 466.73 |

**Table 1.** Overview of our measurements. All values are in ms.

## 8 Non-Blind Signatures

Our Pass2Sign scheme guarantees message blindness towards the server, meaning that the server does not learn the message the device wishes to be signed. This may not always be required or wanted though, e.g., if the message is public or jointly determined or if the server should have control over the messages being signed (for instance because it should also apply throttling based on the message). We therefore sketch in this section a variant Pass2Sign$^*$ of our scheme that does not include message blindness and comes with the additional benefit that the signing protocol is even simpler and verification requires less message pre-processing.

*The Ideal Functionality.* The ideal functionality $\mathcal{F}_{\mathsf{Pass2Sign}^*}$ can be obtained from the one for Pass2Sign by simply including the message $m$ in the output to the server. That is, when the server learns about a signature request, we augment the output to contain the message $m$ provided by the device: in the `<4.Sign Delivery>` interface, the server $\mathcal{S}$ now receives $(\mathsf{SIGNREQ}, sid, qid, status, m)$.

Note that so far the message still remains confidential between $\mathcal{D}$ and $\mathcal{S}$, disregarding the length the message. If the message is supposed to be entirely public, then $m$ must also be included in the output $(\mathsf{SIGNREQ}, sid, qid, \mathcal{D}, m)$ to the adversary $\mathcal{A}$ in the `<3.Sign Request>` interface.

*The Protocol.* We now sketch how our Pass2Sign realization can be modified to one that realizes $\mathcal{F}_{\mathsf{Pass2Sign}^*}$ with entirely public messages. First note that only the signing protocol needs to be changed. In a nutshell, one simply drops all its steps that aim at providing message blindness, such as the hashing of the message including $r$ and $r'$. More precisely, when requesting a signature, the device drops the blinding Step 1b) where $h_m \leftarrow \mathcal{H}(r, m)$ is computed for a random $r$, and the "MAC" Step 1d) which aims at ensuring message consistency without having to store $m$ on the device. Instead, the device now simply keeps the message in its sign record. Depending on whether one aims at the confidential or public message setting, the message is sent either encrypted or in plain to $\mathcal{S}$. For the latter, one has to ensure that the adversary cannot tamper with the message during delivery, and thus we would have to include $m$ in the label of the ciphertext. That is, for the public message setting, the device sends to $\mathcal{S}$ the tuple $(sid, qid, C', m)$ with $C'$ becoming $C' \xleftarrow{r} \mathsf{Enc}(epk, h'_p, (sid, qid, m))$.

The changes to the server's computation are similar: in Step 3b) the server drops its randomness $r'$ and double-hashing contribution and simply signs the (slightly augmented) message $(sid, qid, m)$ based on the received message $m$. The server then returns $(sid, qid, \sigma_{\mathcal{S}})$ to the device.

Finally, in Step 4c), where the device completes the signature to $\sigma_{\mathsf{RSA}}$, the verification of $\sigma_{\mathsf{RSA}}$ is adapted accordingly and, in Step 4d), the full signature is set to $\sigma = (\sigma_{\mathsf{RSA}}, qid)$. Verification of a signature $\sigma$ on message $m$ is simplified to a standard RSA-FDH verification of $\sigma_{\mathsf{RSA}}$ for message $M \leftarrow (sid, qid, m)$.

It is easy to see that the proof of the simplified non-blind scheme $\mathsf{Pass2Sign}^*$ can be derived with minor modifications from the proof of our Pass2Sign scheme. Roughly, we have to adapt the games for the reduction to the RSA assumption, and drop all simulation that stems from the message blinding, such as the simulation of the tags $t$ or the "late-programming" of the random oracle upon a signature verification request (as the messages are now known to the simulator).

## 9 Conclusion

We have introduced a protocol for signing messages with the help of a device and a server. To authenticate towards the server, the user has to enter a password on his device. If the device gets stolen, an adversary is limited to online password guessing attacks, which can be throttled by the server. Neither the device nor

the server are required to be tamper-resistant in any form, yet our protocol offers comparable security to trusted hardware, but without its inconveniences.

The UC-formulation guarantees that our protocol remains secure even in arbitrarily chosen contexts. Our protocol is secure against adaptive corruptions, which properly models our main threat where the device gets lost or stolen. Our model of corruption is even stronger than the existing standard: the simulator does not learn any previous inputs. The ideal functionality also provides a realistic way how password guesses are handled. Namely, even if both the device and the server are corrupted, the adversary does not immediately learn the passwords, but can only mount an offline attack. Thus, if strong passwords are used, the adversary might still not be able to guess them, despite having corrupted both entities.

The protocol is round-optimal and very efficient, as it only requires few random oracle calls and three full-size modular exponentiations for each signature generation. Furthermore, we sketched how to lift the blindness property from our functionality and protocol, yielding an even more efficient scheme.

Possible extensions include achieving full blindness with unlinkability of the resulting signature, an instantiation in the standard model, and extensions to more than one server.

## References


1. J. Algesheimer, J. Camenisch, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO 2002*, pages 417–432, 2002.
2. J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold RSA with adaptive and proactive security. In *EUROCRYPT 2006*, pages 593–611, 2006.
3. A. Bagherzandi, S. Jarecki, N. Saxena, and Y. Lu. Password-protected secret sharing. In *CCS 2011*, pages 433–444, 2011.
4. D. Beaver and S. Haber. Cryptographic protocols provably secure against dynamic adversaries. In *EUROCRYPT 1992*, pages 307–323, 1992.
5. M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, 2003.
6. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS 1993*, pages 62–73, 1993.
7. M. Bellare and R. S. Sandhu. The security of practical two-party RSA signature schemes. *IACR ePrint*, 2001/060, 2001.
8. A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC 2003*, pages 31–46, 2003.
9. D. Boneh, X. Ding, G. Tsudik, and C.-M. Wong. A method for fast revocation of public key certificates and security capabilities. In *USENIX 2001*, 2001.
10. C. Boyd. Digital multisignatures. In *Cryptography and Coding 1989*, pages 241–246, 1989.
11. J. Camenisch, R. R. Enderlein, and G. Neven. Two-server password-authenticated secret sharing uc-secure against transient corruptions. *IACR ePrint*, 2015/006, 2015.
12. J. Camenisch, R. R. Enderlein, and V. Shoup. Practical and employable protocols for uc-secure circuit evaluation over $\mathbb{Z}_n$. In *ESORICS 2013*, pages 19–37, 2013.

13. R. Canetti. Universally composable signature, certification, and authentication. In *CSFW 2004*, pages 219–233, 2004.
14. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT 2005*, pages 404–421, 2005.
15. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/.
16. J.-S. Coron. On the exact security of full domain hash. In *CRYPTO 2000*, pages 229–235, 2000.
17. I. Damgård and G. L. Mikkelsen. On the theory and practice of personal digital signatures. In *PKC 2009*, pages 277–296, 2009.
18. Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *CRYPTO 1989*, pages 307–315, 1989.
19. S. Fehr, D. Hofheinz, E. Kiltz, and H. Wee. Encryption schemes secure against chosen-ciphertext selective opening attacks. In *EUROCRYPT 2010*, pages 381–402, 2010.
20. M. Fischlin. Round-optimal composable blind signatures in the common reference string model. In *CRYPTO 2006*, pages 60–77, 2006.
21. R. Ganesan. Yaksha: augmenting kerberos with pkc. In *NDSS 1995*, pages 132–143, 1995.
22. R. Gennaro, T. Rabin, S. Jarecki, and H. Krawczyk. Robust and efficient sharing of RSA functions. *Journal of Cryptology*, 13(2):273–300, 2000.
23. K. Gjøsteen. Partially blind password-based signatures using elliptic curves. *IACR ePrint*, 2013/472, 2013.
24. K. Gjøsteen and Ø. Thuen. Password-based signatures. In *EuroPKI 2011*, pages 17–33, 2011.
25. J. M. Gosney. Password cracking HPC. Passwords^12 Conference, 2012.
26. C. Hazay, G. L. Mikkelsen, T. Rabin, and T. Toft. Efficient rsa key generation and threshold paillier in the two-party setting. In *CT-RSA 2012*, pages 313–331, 2012.
27. C. Hazay, A. Patra, and B. Warinschi. Selective opening security for receivers. *IACR ePrint*, 2015/860, 2015.
28. Y.-Z. He, C.-K. Wu, and D.-G. Feng. Server-aided digital signature protocol based on password. In *CCST 2005*, pages 89–92, 2005.
29. D. Hofheinz and J. Müller-Quade. Universally composable commitments using random oracles. In *TCC 2004*, pages 58–76, 2004.
30. A. Kiayias and H.-S. Zhou. Equivocal blind signatures and adaptive UC-security. In *TCC 2008*, pages 340–355, 2008.
31. O. Kömmerling and M. G. Kuhn. Design principles for tamper-resistant smartcard processors. In *WOST 1999*, 1999.
32. P. D. MacKenzie and M. K. Reiter. Networked cryptographic devices resilient to capture. *Int. J. Inf. Sec.*, 2(1):1–20, 2003.
33. M. Mannan and P. C. van Oorschot. Using a personal device to strengthen password authentication from an untrusted computer. In *FC 2007*, pages 88–103, 2007.
34. M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *STOC 1990*, pages 427–437, 1990.
35. J. B. Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In *CRYPTO 2002*, pages 111–126, 2002.
36. T. Rabin. A simplified approach to threshold and proactive RSA. In *CRYPTO 1998*, pages 89–104, 1998.

37. M. Venkitasubramaniam. On adaptively secure protocols. In *SCN 2014*, pages 455–475, 2014.
38. S. Xu and R. S. Sandhu. Two efficient and provably secure schemes for server-assisted threshold signatures. In *CT-RSA 2003*, pages 355–372, 2003.

# A Functionalities

Here, we formally introduce the formal UC-definitions for the random oracle functionality $\mathcal{F}_{\mathsf{RO}}^{\mathcal{D}\to\mathcal{R}}$ (Figure 6), the certification authority $\mathcal{F}_{\mathsf{CA}}$ (Figure 7) and the authenticated channel functionality $\mathcal{F}_{\mathsf{Auth}}$ (Figure 8).

1. `Query.` Upon input $(\mathsf{QUERY}, sid, m)$, $m \in \mathcal{D}$, from a party $\mathcal{P}$ do:
   - If there is a tuple $(m, \hat{h}) \in \mathcal{L}$ for some $\hat{h}$, let $h \leftarrow \hat{h}$.
   - Else, draw $\hat{h} \xleftarrow{r} \mathcal{R}$, add $(m, \hat{h})$ to $\mathcal{L}$, and set $h \leftarrow \hat{h}$.
   - Output $(\mathsf{RESPONSE}, sid, m, h)$ to $\mathcal{P}$.

**Fig. 6.** Random Oracle Functionality $\mathcal{F}_{\mathsf{RO}}^{\mathcal{D}\to\mathcal{R}}$, where $\mathcal{L}$ is an initially empty list.

1. `Registration.` Upon input $(\mathsf{REGISTER}, sid, v)$ from a party $\mathcal{P}$:
   - Send $(\mathsf{REGISTER}, sid, v)$ to $\mathcal{A}$. Upon receiving `ok` from $\mathcal{A}$, and if $sid = \mathcal{P}$, and this is the first request from $\mathcal{P}$, record $(sid, v)$.
2. `Retrieve.` Upon receiving a message $(\mathsf{RETRIEVE}, sid)$ from a party $\mathcal{P}'$:
   - Send $(\mathsf{RETRIEVE}, sid, \mathcal{P}')$ to $\mathcal{A}$. Upon receiving `ok` from $\mathcal{A}$:
     - If a record $(sid, v)$ exists, output $(sid, v)$ to $\mathcal{P}'$.
     - Else, output $(sid, \perp)$ to $\mathcal{P}'$.

**Fig. 7.** Certificate Authority Functionality $\mathcal{F}_{\mathsf{CA}}$.

1. `Send.` Upon input $(\mathsf{SEND}, sid, \mathcal{S}, \mathcal{R}, m)$ from a party $\mathcal{S}$:
   - Check that $sid = (\mathcal{S}, \mathcal{R}, sid')$.
   - Send public delayed output $(\mathsf{SEND}, sid, \mathcal{S}, m)$ to $\mathcal{R}$, if $\mathcal{R}$ has not received any output yet.
2. `Corrupt.` Upon receiving a message $(\mathsf{CORRUPT}, sid, m')$ from $\mathcal{A}$:
   - Check that $sid = (\mathcal{S}, \mathcal{R}, sid')$.
   - If $\mathcal{S}$ is not corrupt, ignore.
   - Output $(\mathsf{SEND}, sid, \mathcal{S}, m')$ to $\mathcal{R}$, if $\mathcal{R}$ has not received any output yet.

**Fig. 8.** Authenticated Channel Functionality $\mathcal{F}_{\mathsf{Auth}}$.

**Experiment** $\mathsf{Exp}^{\mathsf{IND-CCA2}}_{\mathsf{ENC},\mathcal{A}}(\tau)$:
$(epk, esk) \xleftarrow{r} \mathsf{EKGen}(1^\tau)$
$b \xleftarrow{r} \{0,1\}$
$((m_0, m_1), \ell, state_{\mathcal{A}}) \xleftarrow{r} \mathcal{A}^{\mathcal{O}^{\mathcal{H}(\cdot)}, \mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}}(epk)$
where $\mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}$ on input $(C_i, \ell_i)$:
return $m'_i \leftarrow \mathsf{Dec}(esk, C_i, \ell_i)$
where $\mathcal{O}^{\mathcal{H}(\cdot)}$ on input $q_j$:
return $h_j \xleftarrow{r} \mathcal{H}(q_j)$
If $|m_0| \neq |m_1| \vee m_0 \notin \mathcal{M} \vee m_1 \notin \mathcal{M}$:
$C \leftarrow \perp$
Else:
$C \xleftarrow{r} \mathsf{Enc}(epk, m_b, \ell)$
$b^* \xleftarrow{r} \mathcal{A}^{\mathcal{O}^{\mathcal{H}(\cdot)}, \mathcal{O}^{\mathsf{Dec}'(\cdot,\cdot)}}(state_{\mathcal{A}}, C)$
where $\mathcal{O}^{\mathsf{Dec}'(\cdot,\cdot)}$ behaves as $\mathcal{O}^{\mathsf{Dec}(\cdot,\cdot)}$,
but returns $\perp$ if $(C, \ell)$ is queried.
return $b^* = b$

**Fig. 9.** Labelled IND-CCA2 for messages with arbitrary length.

## B Proof of Theorem 1 (IND-NC $\implies$ IND-CCA2)

In this section we provide the proof of Theorem 1 that our IND-NC security as specified in Definition 1 implies IND-CCA2 security. For the sake of completeness we first recall the notion of IND-CCA2 security and give our proof afterwards.

**Definition 2 (IND-CCA2 security [34]).** *An encryption scheme* $\mathsf{ENC} = (\mathsf{EKGen}, \mathsf{Enc}, \mathsf{Dec})$ *for arbitrary length messages with labels is IND-CCA2-secure, if for all PPT adversaries* $\mathcal{A}$, $|\Pr[\mathsf{Exp}^{\mathit{IND\text{-}CCA2}}_{\mathsf{ENC},\mathcal{A}}(\tau) = 1] - 1/2| \leq \epsilon(\tau)$ *for some negligible function* $\epsilon$ *and the experiment given in Figure 9.*

*Proof.* Intuitively, the proof formalizes the following idea: if an adversary can decide which message a given ciphertext contains, then it can decide whether a given ciphertext contains any information at all.

Assume an (efficient) adversary $\mathcal{A}$ that guesses the bit $b$ in the IND-CCA2 game with probability at least $\frac{1}{2} + \epsilon$. We can then construct an (efficient) adversary $\mathcal{B}$ such that the probability that it outputs 1 in the two experiments IND-NC-real and IND-NC-ideal differs at least by $\epsilon$. The proof is straightforward. Essentially, the reduction feeds the ciphertexts to $\mathcal{A}$ and if $\mathcal{A}$ guesses correctly, then it must be the real experiments as the ciphertexts in the ideal experiments are unrelated to the challenge message. More precisely, the reduction is as follows. Initially, $\mathcal{B}$ receives a public key $epk$ of the encryption scheme, oracle access to Enc, Dec, and a random oracle $\mathcal{H}$. Thus, $\mathcal{B}$ runs $\mathcal{A}$ on input $epk$. Whenever $\mathcal{A}$ requests access to its decryption oracle, $\mathcal{B}$ forwards $\mathcal{A}$'s query to its own oracle, and returns the unmodified answer to $\mathcal{A}$. When eventually $\mathcal{A}$ outputs the challenge messages $((m_0, m_1), \ell, state_{\mathcal{A}})$, $\mathcal{B}$ checks that $|m_0| = |m_1|$, and that both

$m_0 \in \mathcal{M}$ and $m_1 \in \mathcal{M}$. If any checks fail, $\mathcal{B}$ sets $c \leftarrow \perp$. Else, $\mathcal{B}$ picks a random bit $b \in \{0,1\}$ and calls its own encryption oracle: $c \leftarrow \mathsf{Enc}(m_b, \ell)$. $\mathcal{B}$ then passes $(state_{\mathcal{A}}, c)$ to $\mathcal{A}$. If now $(c, \ell)$ is queried to the decryption oracle, $\mathcal{B}$ responds with $\perp$. For every other query, $\mathcal{B}$ uses its own decryption oracle, and forwards the answer to $\mathcal{A}$. Eventually, $\mathcal{A}$ outputs its guess $b^*$. $\mathcal{B}$ outputs 1 if $b^* = b$, and 0 otherwise. Let us analyze the probability that $\mathcal{B}$ outputs 1 in both experiments.

1. If $\mathcal{B}$ is run in the experiment $\mathsf{IND\text{-}NC\text{-}real}$, then clearly, $\mathcal{B}$ was playing the IND-CCA2 game with $\mathcal{A}$, using the random bit $b$. Hence, we have $\Pr[\mathsf{Exp}^{\mathsf{IND\text{-}NC\text{-}real}}_{\mathsf{NCE},\mathcal{B}} = 1] = \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}CCA2}}_{\mathsf{NCE},\mathcal{A}} = b'] = \frac{1}{2} + \epsilon$.
2. If $\mathcal{B}$ run in experiments $\mathsf{IND\text{-}NC\text{-}ideal}$, then the challenge ciphertext that $\mathcal{B}$ fed to $\mathcal{A}$ was unrelated to $m_b$, i.e., the answer $b^*$ of $\mathcal{A}$ is information-theoretically independent of $b$ and therefore $\Pr[\mathsf{Exp}^{\mathsf{IND\text{-}NC\text{-}ideal}}_{\mathsf{NCE},\mathcal{B}} = 1] = \frac{1}{2}$ follows.

So, the difference between these two probabilities is at least $\epsilon$.

## C Proof of Theorem 2 (IND-NC Security of BR-Encryption)

This section is devoted to prove Theorem 2, i.e., showing that the modified Bellare-Rogaway encryption as described in Section 3 achieves our notion of IND-NC security in the random-oracle model.

*Proof.* The proof is related to the one given by Nielsen [35] for his construction; we also reduce the security of our construction to the one-wayness of the underlying trapdoor permutation. For the sake of readability, we first describe our simulator $\mathsf{SIM}_{\mathsf{NCE}}$, and then prove via a sequence of games that our scheme with the given simulator is IND-NC secure, as required by Definition 1. The simulator processes its tasks as follows.

**Key Generation.** On input of $(\mathsf{publickey}, 1^\tau)$, $\mathsf{SIM}_{\mathsf{NCE}}$ generates $(f, f^{-1}, \Sigma) \xleftarrow{r} \mathsf{TFGen}_{\mathsf{f}}(1^\tau)$. It stores $esk = f^{-1}$, and returns $(f, \Sigma)$ as $epk$.

**Encryption.** On input of $(\mathsf{encrypt}, k, \ell)$, $\mathsf{SIM}_{\mathsf{NCE}}$ draws $r \xleftarrow{r} \Sigma$. It also saves the randomness $r$ for further usage. Namely, if the same $r$ was drawn before, the simulator aborts. Let $l \leftarrow |\mathsf{ec}(1^k)|/\tau$.[1] Set $C_1 \leftarrow f(r)$, $C_2^i \xleftarrow{r} \{0,1\}^\tau$, for all $0 < i \leq l$, and $C_3 \xleftarrow{r} \{0,1\}^\tau$. $\mathsf{SIM}_{\mathsf{NCE}}$ returns $C \leftarrow (C_1, (C_2^1, C_2^2, \ldots, C_2^l), C_3)$.

**RO Queries.** With the input of $(\mathsf{roquery}, q)$ both random oracles $\mathcal{G}$ and $\mathcal{K}$ are addressed, i.e., we assume that $q$ is prefixed accordingly (e.g., $q = (\mathcal{G}, q')$). If the input does not satisfy this format, $\mathsf{SIM}_{\mathsf{NCE}}$ ignores the inputs (in the real world, the same behavior is enforced by definition). We use $\mathcal{L}_{\mathcal{G}}$ and $\mathcal{L}_{\mathcal{K}}$ as a short-hand notation to refer to the lists of queries and answers to $\mathcal{G}$ and $\mathcal{K}$, respectively. If for any of the random-oracle queries the simulator draws a response twice, it aborts.

**Decryption.** On input of $(\mathsf{decrypt}, (C_1, (C_2^1, C_2^2, \ldots, C_2^k), C_3), \ell)$, $\mathsf{SIM}_{\mathsf{NCE}}$ checks if a preimage $(r', k, m', \ell)$ of $C_3$ for $\mathcal{K}$ and images of $(i, r')$ for $\mathcal{G}$ and all

[1] Note, this is always an integer.

$1 < i \leq k$ have been defined. Let $(m'_1, m'_2, \ldots, m'_{k'}) \leftarrow \mathsf{ec}(m')$. If $k \neq k'$, the simulator returns $\perp$. It then checks whether $C_2^i = \mathcal{G}(i, r') \oplus m'_i$ for all $1 < i \leq k$, and $C_1 = f(r')$ holds. If so $\mathsf{SIM_{NCE}}$ returns $m'$. Otherwise, the simulator makes the missing oracle calls. If these make the ciphertext valid, the simulator aborts, otherwise it returns $\perp$.

**Key Leakage.** On input of $(\mathsf{keyleak}, \mathcal{Q})$, $\mathsf{SIM_{NCE}}$ must program the random oracles to ensure consistent decryption of ciphertexts as the adversary will receive the secret key $esk$. That is, for all $C_j = (C_{1,j}, (C_{2,j}^1, C_{2,j}^2, \ldots, C_{2,j}^{k_j}), C_{3,j})$, and $m_j \in \mathcal{Q}$, it programs $\mathcal{G}$ and $\mathcal{K}$ as follows: Let $(m_{1,j}, \ldots m_{k_j,j}) \leftarrow \mathsf{ec}(m_j)$. $\mathsf{SIM_{NCE}}$ adds $((i, (f^{-1}(C_{1,j}))), m_{i,j} \oplus C_{2,j}^i)$ for $0 < i \leq k_j$ to $\mathcal{L}_\mathcal{G}$ and adds $((f^{-1}(C_{1,j}), k_j, m_j, \ell_j), C_{3,j})$ to $\mathcal{L}_\mathcal{K}$. Finally, $\mathsf{SIM_{NCE}}$ returns the secret key $esk$, i.e., $f^{-1}$. If the programming fails at some point, i.e., a preimage already exists, the simulator aborts.

We now show that the simulator is such that the adversary will output 1 in both experiments with essentially the same probability. We denote by $\mathcal{P}_i$ the probability that $\mathcal{A}$ outputs 1 in Game $i$.

Game 0: In the first game, the real protocol is run with the adversary, i.e., $\mathsf{IND\text{-}NC\text{-}real}$. Hence, we have $\Pr[\mathsf{Exp}^{\mathsf{IND\text{-}NC\text{-}real}}_{\mathsf{NCE},\mathcal{A}}(1^\tau) = 1] = \mathcal{P}_0$.

Game 1: Next, each query $(m_i, \ell_i)$ to $\mathcal{O}^{\mathsf{Enc}}$ and the corresponding answer $C_i$ is stored in a list $\mathcal{L}_{\mathsf{Enc}}$. In particular, it is encrypted honestly, but $((m_i, \ell_i), C_i)$ is stored in a list $\mathcal{L}_{\mathsf{Enc}}$. Whenever the adversary queries the decryption oracle with $(C_i, \ell_i)$, $m_i$ is returned, if an entry $((m_i, \ell_i), C_i) \in \mathcal{L}_{\mathsf{Enc}}$ for some $m_i$ exists. In other words, such ciphertexts are not decrypted anymore. This is only an internal change, due to the perfect correctness of our scheme and thus we have $\left|\mathcal{P}_0 - \mathcal{P}_1\right| = 0$.

Game 2: We now start gradually building the simulator $\mathsf{SIM_{NCE}}$. In the first step, the simulator $\mathsf{SIM_{NCE}}$ receives control of the random oracles $\mathcal{G}$ and $\mathcal{K}$. We abort, if the simulator draws a response twice. This only happens with probability $\frac{q_h^2}{2^\tau}$ due to the birthday paradox, where $q_h$ is the number of random oracle queries made. $\left|\mathcal{P}_1 - \mathcal{P}_2\right| \leq \frac{q_h^2}{2^\tau}$ follows.

Game 3: Next, the simulator $\mathsf{SIM_{NCE}}$ generates the key pair. In particular, on input of $(\mathsf{publickey}, 1^\tau)$, it generates $(epk, esk) \xleftarrow{r} \mathsf{EKGen}(1^\tau)$, and returns $(epk, esk)$. This step is purely conceptional, so we have $\left|\mathcal{P}_2 - \mathcal{P}_3\right| = 0$.

Game 4: Now, the encryption oracle $\mathcal{O}^{\mathsf{Enc}}$ is simulated by $\mathsf{SIM_{NCE}}$. In particular, on input of $(\mathsf{encrypt}, m, \ell)$ (the simulator at this point still receives the full message $m$), the simulator draws $r \xleftarrow{r} \Sigma$, calculates $C \xleftarrow{r} \mathsf{Enc}(esk, m, \ell; r)$, and returns $(r, C)$. Note, $\Sigma$ is exponential in size. However, the simulator aborts, if it draws a randomness $r_i$ a twice. This only happens with probability at most

$\frac{q^2}{|\Sigma|}$ due to the birthday paradox, where $q$ is the number of queries made to $\mathsf{Enc}$. Hence, we have $\left|\mathcal{P}_3 - \mathcal{P}_4\right| \leq \frac{q^2}{|\Sigma|}$.

GAME 5: Next, the simulator $\mathsf{SIM}_{\mathsf{NCE}}$ takes over the decryption of cipher-texts. On input of $(\mathsf{decrypt}, C, \ell)$, $\mathsf{SIM}_{\mathsf{NCE}}$ runs the decryption algorithms and returns the results. This is only a conceptual change, so $\left|\mathcal{P}_4 - \mathcal{P}_5\right| = 0$.

GAME 6: The simulator $\mathsf{SIM}_{\mathsf{NCE}}$ now no longer returns *esk* directly after creation, but only when it is queried with $(\mathsf{keyleak}, \cdot)$, which is sent once the adversary is finished with its query phase. As *esk* is not required in the meantime, this is a conceptional change only, thus $\left|\mathcal{P}_5 - \mathcal{P}_6\right| = 0$.

GAME 7: The simulator now always returns $\perp$ whenever the adversary queries the decryption oracle with a ciphertext $C$ for which not all random oracle calls have been made. In particular, the simulator checks whether a preimage $(r', k, m, \ell)$ of $C_3$ for $\mathcal{K}$, and images of $(i, r')$ for $\mathcal{G}$ and all $1 < i \leq k$ have been defined and whether $C_2^i = \mathcal{G}(i, r') \oplus m_i$ for all $1 < i \leq k$ and $C_1 = f(r')$ holds, while $k = |\mathsf{ec}(1^k)|/\tau$ must hold. If any of these checks fail, the simulator outputs $\perp$. Note, the preimages are unique if they exist. Also, the simulator makes the missing random oracle calls. If these turn the ciphertext into a valid one, the simulator aborts. The adversary will only notice a difference to the Game 6 if the latter happens, which is with probability at most $\frac{q_h q_d}{2^\tau}$, where $q_h$ is the number of random oracle queries made and $q_d$ denotes the number of calls to the decryption oracle. Hence, we have $\left|\mathcal{P}_6 - \mathcal{P}_7\right| \leq \frac{q_h q_d}{2^\tau}$.

GAME 8: We now gradually change the simulator's input; instead of giving $\mathsf{SIM}_{\mathsf{NCE}}$ the message $m$ before it has to come up with an encryption $C$, the challenger provides $m$ after the adversary is done with its query phase, i.e., when the simulator receives $(\mathsf{keyleak}, \mathcal{Q})$. In particular, the simulator $\mathsf{SIM}_{\mathsf{NCE}}$ only receives $(\mathsf{encrypt}, |m|, \ell)$ instead of $(\mathsf{encrypt}, m, \ell)$. Eventually, $(\mathsf{keyleak}, \mathcal{Q})$ (containing the messages the cipher-texts should contain) is sent to $\mathsf{SIM}_{\mathsf{NCE}}$, which allows $\mathsf{SIM}_{\mathsf{NCE}}$ to program the random oracles accordingly. We prove that this only affects the view of the adversary negligibly by a series of hybrids. Let $q$ be an upper bound of queries to the encryption oracle.

HYBRID 8.$j$: Up to encryption query $j$, $0 \leq j \leq q$, $\mathsf{SIM}_{\mathsf{NCE}}$ is given $(\mathsf{encrypt}, |m|, \ell)$ instead of $(\mathsf{encrypt}, m, \ell)$. So, Game 8.0 is identical to Game 7. On inputs $(\mathsf{encrypt}, m, \ell)$, the simulator behaves as before. Now we describe how the simulator $\mathsf{SIM}_{\mathsf{NCE}}$ proceeds on inputs $(\mathsf{encrypt}, |m|, \ell)$. It draws $r \xleftarrow{\text{r}} \Sigma$ as in $\mathsf{Enc}$ and sets $C_1 \leftarrow f(r)$. Let $l \leftarrow |\mathsf{ec}(1^{|m|})|/\tau$. For each $0 < i \leq l$, $\mathsf{SIM}_{\mathsf{NCE}}$ draws $C_2^i \xleftarrow{\text{r}} \{0,1\}^\tau$. Finally, it draws $C_3 \xleftarrow{\text{r}} \{0,1\}^\tau$ and returns $C \leftarrow (C_1, C_2, C_3)$. Note, the distributions of $C$ are exactly the same as honest encryptions for the same message length $|m|$.

Eventually, $\mathsf{SIM}_{\mathsf{NCE}}$ is queried $(\mathsf{keyleak}, \mathcal{Q})$ and thus receives the message $m$ corresponding to each $(\mathsf{encrypt}, |m|, \ell)$ call made earlier. To achieve consistent

decryption, the simulator now programs the random oracles $\mathcal{K}$ and $\mathcal{G}$, such that a decryption of each generated $C$ (with the correct label $\ell$) returns the corresponding $m$. Let $(m_1, m_2, \ldots, m_k) \leftarrow \mathsf{ec}(m)$. $\mathsf{SIM}_{\mathsf{NCE}}$ adds $((i,r), m_i \oplus C_2^i)$ for each block $m_i$, $0 < i \leq k$ in $m$ to $\mathcal{L}_{\mathcal{G}}$. Additionally, it adds $((r,k,m,\ell), C_3)$ to $\mathcal{L}_{\mathcal{K}}$. This programming may fail if one of these preimages already exists in $\mathcal{L}_{\mathcal{G}}$ or $\mathcal{L}_{\mathcal{K}}$. This is only possible if the adversary had already queried $(r,k,m,\ell)$ to $\mathcal{K}$ or $(i,r)$ for some $i$, $0 < i \leq k$ to $\mathcal{G}$, as $\mathsf{SIM}_{\mathsf{NCE}}$ always draws fresh random coins.

Reduction: Assuming that the adversary can distinguish between Hybrid 8.$j$ and Hybrid 8.$j+1$, we can turn the adversary $\mathcal{A}$ into an adversary $\mathcal{B}$ which inverts a given element $c$, i.e., outputs $f^{-1}(c)$ with non-negligible probability. To do so, adversary $\mathcal{B}$ receives $c$ and the corresponding parameters $\Sigma$ and $f$ from the trapdoor game. It embeds the challenge $c$ for $\mathcal{A}$ as follows. It draws a random index $j \xleftarrow{r} \{1, 2, \ldots, l\}$ and then on the $j$th query to the encryption oracle, $C_1$ is not calculated from a honestly drawn $r$, but is set to the provided challenge $c$, i.e., $C_1 \leftarrow c$. Every other query is processed as in the prior game. In particular, we can extract $r = f^{-1}(C_1)$ from $\mathcal{L}_{\mathcal{G}}$ or $\mathcal{L}_{\mathcal{K}}$ resp., and can therefore simulate the decryption oracle, if queried with a correctly computed ciphertext. Note that other ciphertexts are already excluded. Hence, the embedding does not change the view of the adversary. Assuming that we cannot program the random oracles accordingly, i.e., the adversary notices a difference to the prior game, then the adversary must have made a query $(r', k', m', \ell')$ to the random oracle $\mathcal{K}$, such that $c = C_1 = f(r')$, or a query $(i, r')$ for some $i$, $0 < i \leq k$, to $\mathcal{G}$ for some $C = (C_1, C_2, C_3)$ returned by $\mathsf{SIM}_{\mathsf{NCE}}$. Let the probability of this event be $\epsilon_l$. We can now derive that $\mathcal{B}$ can invert $f$ with probability $\frac{\epsilon_l}{l}$, which is non-negligible, if $\epsilon_l$ is non-negligible. This is a contradiction to the assumption that the element $c$ cannot be inverted with noticeable probability. $\left|\mathcal{P}_7 - \mathcal{P}_8\right| \leq \sum_{i=0}^{q} \epsilon_i$ follows.[2]

Game 9: Finally, IND-NC-ideal is run with the simulator given in Game 8. This is only an internal change: the adversary does not note any difference. $\left|\mathcal{P}_8 - \mathcal{P}_9\right| = 0$ follows.

As an upper bound, we can therefore derive $\left|\mathcal{P}_0 - \mathcal{P}_9\right| \leq \sum_{i=1}^{9} \left|\mathcal{P}_{i-1} - \mathcal{P}_i\right|$, i.e., $\left|\Pr[\mathsf{Exp}^{\mathsf{IND\text{-}NC\text{-}real}}_{\mathsf{NCE},\mathcal{A}}(\tau) = 1] - \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}NC\text{-}ideal}}_{\mathsf{NCE},\mathcal{A},\mathsf{SIM}_{\mathsf{NCE}}}(\tau) = 1]\right|$ is negligible.

# D Proof of Theorem 3 (Security of our Pass2Sign Protocol)

We now prove that our Pass2Sign protocol described in Section 5 indeed satisfies the ideal functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$ defined in Section 4.

## D.1 Sequence of Games

Our proof consist of a sequence of games that a challenger runs with the real-world adversary. In our final game we then make the transition to let the chal-

[2] $q$ is at most polynomial, therefore the given sum is also negligible.

lenger run internally the ideal functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$ and simulate all messages based merely on the information it can obtain from $\mathcal{F}_{\mathsf{Pass2Sign}}$. We now describe each game $i$ and argue why the view of the environment does not significantly change.

GAME 0: In the first game, the challenger executes the real protocol for *all* honest players, obtaining their inputs from, and passing the respective outputs to the environment.

GAME 1: Let $\mathcal{T}_{\mathcal{H}}$ be the table in which the simulator stores query-response pairs $(m, h)$ when simulating random-oracle queries to $\mathcal{H}$. Recall that all random oracle calls to $\mathcal{H}$ are implicitly prefixed with *sid* of the current account. Similarly, we create a table $\mathcal{T}_{\mathcal{H}}$ per *sid*, but will omit the explicit handling of the *sid* here as well. This game aborts whenever the adversary or the challenger cause a collision in $\mathcal{H}$ or $\mathcal{H}_{\mathsf{RSA}}$, or the adversary sends a "correctly predicted" hash value. A "correctly predicted" value is some hash $h_p, h'_p, h_m$ that neither resulted from a query to the random oracle nor was previously generated by the challenger, and the adversary makes a random-oracle query that maps to this value only after having sent the hash.

By the random choice of the response from $\{0,1\}^\tau$, the adversary can distinguish this game hop only with negligible probability.

GAME 2: From now on, the challenger creates additional internal records for setup and signing. When setup is done by an honest device, it stores $(\mathsf{setup\text{-}sim}, sid, k, h_p, d_{\mathcal{S}}, d_{\mathcal{D}}, (N, e))$, i.e., it also keeps the RSA key share $d_{\mathcal{S}}$ of the server. For an account created by a corrupt device with an honest server, the challenger maintains $(\mathsf{setup\text{-}sim}, sid, \perp_k, h_p, d_{\mathcal{S}}, \perp_{d_{\mathcal{D}}}, (N, e))$.

Similarly, when an honest device starts a signing request, the challenger initiates a record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, m_i, t_i, r_i, \perp_{r'}, \perp_\sigma)$. For a signing session between a corrupt device and honest server, the challenger creates $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \perp_m, t_i, \perp_r, r', \perp_\sigma)$. Both records will be completed with the missing values as soon as they are generated or received by an honest party.

Clearly, this is only an internal change and has no effect on the view of the environment.

GAME 3: In this game, we replace every non-committing encryption (via a series of hybrids) that would be sent between two honest parties by a simulated ciphertext. More precisely, we first deploy the simulator $\mathsf{SIM}_{\mathsf{NCE}}$ of the non-committing encryption scheme in mode $\mathsf{SIM}_{\mathsf{NCE}}(\mathsf{publickey}, 1^\tau)$ to obtain the public key *epk* that the honest server registers with $\mathcal{F}_{\mathsf{CA}}$. Then, whenever the honest device has to create an encryption of $m$ with label $\ell$ under *epk* we replace the real ciphertext by $C \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{encrypt}, |m|, \ell)$ where $|m|$ denotes the message length. We also internally maintain a list $\mathcal{Q}$ of tuples $(C, m, \ell)$ mapping the real messages to the "dummy" ciphertexts.

When an honest server receives such a simulated ciphertext, it does not decrypt $C$ but looks up the corresponding plaintext from its internal record, i.e., either $\mathsf{setup\text{-}sim}$ for ciphertexts received in setup and $\mathsf{sign\text{-}sim}$ for ciphertexts received in the signing protocol. If the server receives a ciphertext/label pair $(C', \ell')$ where $(C', \cdot, \ell') \notin \mathcal{Q}$, i.e., that was not created by an honest device, we run $\mathsf{SIM}_{\mathsf{NCE}}(\mathsf{decrypt}, C', \ell')$ instead of the decryption algorithm.

When the server eventually gets corrupted, we finally deploy $\mathsf{SIM}_{\mathsf{NCE}}(\mathsf{keyleak}, \mathcal{Q})$ to obtain the secret decryption key $esk$ we have to hand to the adversary.

This game hop is indistinguishable by the IND-NC security of the encryption scheme $(\mathsf{EKGen}, \mathsf{Enc}, \mathsf{Dec})$.

Game 4: Here, we abort if an honest party obtains an input $(\mathsf{VERIFY}, sid, m^*, \sigma^*, pk)$ for a public key $pk$ of an honest device, but for a signature $\sigma^*$ that the device has never produced. We show that receiving such a forged signature in our protocol allows to construct an adversary $\mathcal{B}$ that breaks the strong unforgeability of RSA-FDH signatures with non-negligible probability. For the sake of simplicity, we use the same reduction for the case of an honest and a corrupt server.

When $\mathcal{B}$ receives the challenge RSA public key $pk = (N, e)$, we choose $d_{\mathcal{S}}$ at random from $\mathbb{Z}_N$, leave $d_{\mathcal{D}}$ unassigned and normally compute the authentication values for our protocol. Recall that the challenger also internally stores $d_{\mathcal{S}}$. For each signing query $m_i$, the honest device chooses random $r_i$ and $h_{m_i}$, gets $h_{m_i} \leftarrow \mathcal{H}(r_i, m_i)$ from the random oracle, and sends $h_{m_i}$ to the server. When it receives a response $(sid, qid_i, h_{m,i}, t_i, r'_i, \sigma_{\mathcal{S},i})$, $\mathcal{B}$ verifies that $\sigma_{\mathcal{S},i} = \mathcal{H}_{\mathsf{RSA}}(sid, qid_i, \mathcal{H}(r'_i, h_{m_i}))^{d_{\mathcal{S}}}$ and aborts otherwise. Then $\mathcal{B}$ sends $M_i \leftarrow (sid, qid_i, \mathcal{H}(r'_i, h_{m_i}))$ to its RSA-signing oracle to obtain the signature $\sigma_{\mathsf{RSA},i}$. The device outputs $(\mathsf{SIGNATURE}, sid, qid_i, \sigma_i)$, where $\sigma_i \leftarrow (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$. We store each produced message/signature tuple $(m_i, \sigma_i)$ in a list $\mathcal{L}_{\mathsf{Sign}}$.

When $\mathcal{B}$ receives an input $(\mathsf{VERIFY}, sid, m^*, \sigma^*, pk)$ such that $(m^*, \sigma^*) \notin \mathcal{L}_{\mathsf{Sign}}$, it parses $\sigma^*$ as $(\sigma^*_{\mathsf{RSA}}, qid^*, r^*, r'^*)$ and outputs $(M^*, \sigma^*_{\mathsf{RSA}})$ as forgery with $M^* \leftarrow (sid, qid^*, \mathcal{H}(r'^*, \mathcal{H}(r^*, m^*)))$. The output is a valid RSA-FDH forgery if the message/signature tuple $(M^*, \sigma^*_{\mathsf{RSA}})$ did not appear as an RSA signing query/response before. Suppose for contradiction that it did appear in a signing query by $\mathcal{B}$, i.e., $M^* = (sid, qid^*, \mathcal{H}(r'^*, \mathcal{H}(r^*, m^*))) = M_i = (sid, qid_i, \mathcal{H}(r'_i, \mathcal{H}(r_i, m_i)))$ for some query $i$ that led to signature $\sigma^*_{\mathsf{RSA}}$. That would mean that $qid^* = qid_i$ and $\mathcal{H}(r'^*, \mathcal{H}(r^*, m^*)) = \mathcal{H}(r'_i, \mathcal{H}(r_i, m_i))$. As we excluded collisions on $\mathcal{H}$ in Game 1, the latter means that $r'^* = r'_i$, $r^* = r_i$, and $m^* = m_i$. This contradicts the fact that $(m^*, \sigma^*) \notin \mathcal{L}_{\mathsf{Sign}}$.

Note that our simulation deviated from how the signing key $d_{\mathcal{S}}$ for the server was chosen. In the original scheme, $d_{\mathcal{S}}$ is chosen from $\mathbb{Z}^*_{\varphi(N)}$, whereas we selected $d_{\mathcal{S}}$ randomly from $\mathbb{Z}_N$, since the challenger only learns $(N, e)$. We use Lemma 5.1 from [7] here where it was shown that $\Pr[d_{\mathcal{S}} \in \mathbb{Z}^*_{\varphi(N)}] > \frac{8}{435 \ln |N|}$ for RSA modulus $N$ and $d_{\mathcal{S}} \xleftarrow{r} \mathbb{Z}_N$. Hence, we have chosen a "good" $d_{\mathcal{S}}$ value with non-negligible probability, in which case the simulation of our protocol was perfect.

Overall, this game hop is indistinguishable by the strong unforgeability of the signature scheme $(\mathsf{SKGen}_{\mathsf{RSA}}, \mathsf{Sign}_{\mathsf{RSA}}, \mathsf{Verify}_{\mathsf{RSA}})$.

GAME 5: In this game we consider the setting where the server is honest and the device is corrupt, but was honest during setup. Then, similar to the previous game, we will abort if we see a forged signature for the corresponding key $pk$. That is, we abort when we see a valid signature $(m^*, \sigma^*)$ where $\sigma^* = (\sigma^*_{\mathsf{RSA}}, qid^*, r^*, r'^*)$ but the honest server never signed $M^* = (sid, qid^*, h'_m)$ where $h'^*_m = \mathcal{H}(r'^*, \mathcal{H}(r^*, m^*))$ in the session $qid^*$, or when we see two different valid message-signature pairs $(m^*_1, \sigma^*_1) \neq (m^*_2, \sigma^*_2)$ that resulted from the same signing protocol for $M^* = (sid, qid^*, h'_m)$. In contrast to the previous game, we cannot use the unforgeability of the RSA signature as underlying assumption, though, as the honest server does not have "full" control of the final signature. However, we can show that a forgery in our scheme allows to break the RSA problem in the random-oracle model.

The latter case of two different message-signature pairs either requires the adversary to find collisions in the random-oracle responses for $\mathcal{H}$ and $\mathcal{H}_{\mathsf{RSA}}$, which we excluded in GAME 1, or to find two different values $0 < \sigma^*_{\mathsf{RSA},1} < \sigma^*_{\mathsf{RSA},2} < N$ such that $\mathcal{H}_{\mathsf{RSA}}(M^*) \equiv {\sigma^*_{\mathsf{RSA},1}}^e \equiv {\sigma^*_{\mathsf{RSA},2}}^e \mod N$. If $\mathcal{H}_{\mathsf{RSA}}(M^*) \in \mathbb{Z}^*_N$, then this is impossible because RSA is a permutation on $\mathbb{Z}^*_N$. If $\mathcal{H}_{\mathsf{RSA}}(M^*) \notin \mathbb{Z}^*_N$, then $\gcd(N, \mathcal{H}_{\mathsf{RSA}}(M^*)) > 1$, so that $\mathcal{B}$ can factor $N$ and solve the RSA problem.

For the former case of a (non-strong) forgery against our protocol, consider algorithm $\mathcal{B}$ that, on input an RSA public key $N, e$ and challenge $y \in \mathbb{Z}^*_N$, outputs $x$ where $x^e \equiv y \mod N$ with non-negligible probability. $\mathcal{B}$ chooses keys $d_{\mathcal{D}} \xleftarrow{r} \mathbb{Z}_N$ and $k \xleftarrow{r} \{0,1\}^\tau$. When the device gets corrupted, $\mathcal{B}$ hands these values to the adversary. When the adversary makes a random-oracle query $\mathcal{H}_{\mathsf{RSA}}(M_i)$, $\mathcal{B}$ chooses $x_i \xleftarrow{r} \mathbb{Z}^*_N$ and responds $x^e_i y \mod N$. When the honest server receives an incoming signing request for message hash $h_{m_i}$, it chooses $r'_i \xleftarrow{r} \{0,1\}^\tau$, $\sigma_{\mathcal{S}_i} \xleftarrow{r} \mathbb{Z}^*_N$, programs $\mathcal{H}_{\mathsf{RSA}}(sid, qid_i, \mathcal{H}(r'_i, h_{m_i})) = \sigma^{e d_{\mathcal{D}}}_{\mathcal{S}_i} \mod N$, adds $(h_{m_i}, \sigma^{d_{\mathcal{D}}}_{\mathcal{S}_i})$ to $\mathcal{L}_{\mathsf{Sign}}$, and sends $\sigma_{\mathcal{S}_i}$ back to the device. In case $\mathcal{H}_{\mathsf{RSA}}(sid, qid_i, \mathcal{H}(r'_i, h_{m_i}))$ had been queried before, $\mathcal{B}$ aborts, but by the random choice of $r'_i$, this happens with probability at most $\frac{q_s q_h}{2^\tau}$, where $q_s$ and $q_h$ are the number of signing and random-oracle queries made by the adversary, respectively. When $\mathcal{B}$ receives an input $(\mathsf{VERIFY}, sid, m^*, (\sigma^*_{\mathsf{RSA}}, qid^*, r^*, r'^*), pk)$ such that $(\mathcal{H}(r^*, m^*), \sigma^*) \notin \mathcal{L}_{\mathsf{Sign}}$, $\mathcal{B}$ looks in its random-oracle responses to find $x^*$ such that $\mathcal{H}_{\mathsf{RSA}}(sid, qid^*, \mathcal{H}(r'^*, \mathcal{H}(r^*, m^*))) = x^{*e} y \mod N$, then by the validity of the signature we have that ${\sigma^*_{\mathsf{RSA}}}^e \equiv x^{*e} y \mod N$, so $\mathcal{B}$ returns $\sigma^*_{\mathsf{RSA}}/x^* \mod N$ as the RSA inversion of $y$. As in the previous game, we use Lemma 5.1 from [7] for a lower bound on the probability that $d_{\mathcal{D}} \in \mathbb{Z}^*_{\varphi(N)}$.

GAME 6: We now make all values that would depend on the device secret $k$ independent of $k$, when they are generated by an honest device and are sent to a corrupt server. More precisely, we change the way the authentication values $h_p, h'_{p,i}$ and the tags $t_i$ are computed. Namely, we choose $h_p \xleftarrow{r} \{0,1\}^\tau$ at setup, and store the actual password $pwd$ in an internal record $(\mathsf{pwdsetup}, sid, pwd)$.

For a sign request started by an honest device we derive $h'_{p,i} \leftarrow \mathcal{H}(qid_i, h_p)$ if the request was initiated for a password $pwd'_i = pwd$ and set it to a random value $h'_{p,i} \xleftarrow{r} \{0,1\}^\tau$ if the passwords did not match. We keep the created $h'_{p,i}$ in the sign-sim record as before, and also create another record for the password attempts as $(\mathsf{pwdsign}, sid, qid_i, pwd'_i)$.

The computation of the tag $t_i \leftarrow \mathcal{H}(\texttt{"MAC"}, qid_i, k, h_{m,i})$ is replaced by $t_i \xleftarrow{r} \{0,1\}^\tau$, and $t_i$ is stored in sign-sim as before. We then also replace the verification of the tag in Step 4 of the signing protocol by a simple look up whether the received $t_i$ value is the same as in sign-sim.

If we receive a random oracle query of the form $\mathcal{H}(k, \cdot)$ or $\mathcal{H}(\texttt{"MAC"}, \cdot, k, \cdot)$ for the secret key $k$ that we have chosen and the device is still honest, we abort. By the random choice of $k \xleftarrow{r} \{0,1\}^\tau$, and the fact that we just made all values independent of $k$, the probability for such a query (and the resulting abort) is negligible.

However, we change that behavior as soon as the device gets corrupted, as then the adversary learns $k$ and the above argument no longer holds. With the help of our internal records, we can choose a fresh $k$ at the moment the device gets corrupted and ensure consistency with the previously chosen values $h_p, h'_{p,i}$ and $t_i$ as follows. For a query $\mathcal{H}(k, pwd)$ where a record $(\mathsf{setup\text{-}sim}, sid, k, h_p, d_\mathcal{S}, d_\mathcal{D}, (N, e))$ for $k$ and a record $(\mathsf{pwdsetup}, sid, pwd)$ for $sid, pwd$ exists, we set the random oracle response to be $h_p$ (taken from the setup-sim record). Similarly, for a query $\mathcal{H}(qid_i, h^*_p)$ where $((k, pwd'_i), h^*_p) \in \mathcal{T}_\mathcal{H}$, i.e., $h^*_p$ is the result of a previous query $(k, pwd'_i)$ and records $(\mathsf{setup\text{-}sim}, sid, k, h_p, d_\mathcal{S}, d_\mathcal{D}, (N, e))$ for $k$, and $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, m_i, t_i, r_i, \{r'_i, \perp_{r'}\}, \{\sigma_{\mathsf{RSA},i}, \perp_\sigma\})$ and $(\mathsf{pwdsign}, sid, qid_i, pwd'_i)$ for $sid, qid_i, pwd'_i$ exists, we set the response to $\mathcal{H}(qid_i, h^*_p) \leftarrow h'_{p,i}$ (taken from sign-sim) and to a random value otherwise. For queries $\mathcal{H}(\texttt{"MAC"}, qid_i, k, h_{m,i})$ where a setup-sim record for $k$ and a record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, m_i, t_i, r_i, \{r'_i, \perp_{r'}\}, \{\sigma_{\mathsf{RSA},i}, \perp_\sigma\})$ for $sid, qid_i, h_{m,i}$ exists, we set the random oracle response to the stored $t_i$ and to a random string otherwise.

Overall, this game hop is indistinguishable with overwhelming probability by the random choice of $k$.

Game 7: We now make a similar change to the computation of $h_p$ and $h'_{p,i}, t_i$ for the setting where a setup or sign request is done between an honest device and *honest* server. However, here we leave all values $h_p$ and $h'_{p,i}, t_i$ unassigned at the beginning and only create records $(\mathsf{pwdsetup}, sid, pwd)$ for setup and $(\mathsf{pwdsign}, sid, qid_i, pwd'_i)$ for each signing request.

When the honest server is supposed to output $(\mathsf{SIGNREQ}, sid, qid_i, c_i)$ where $c_i$ indicates whether the password attempt was successful, it determines $c_i$ based on the passwords stored in the pwdsign and pwdsetup records.

Then, if the server gets the ok to proceed and the password attempt was correct, i.e., $c_i = \mathsf{pwdok}$ we choose a random tag $t_i \xleftarrow{r} \{0,1\}^\tau$ and store it in the sign-sim record. The honest server then proceeds normally. When the honest device receives a message $(sid, qid_i, h_{m,i}, t_i, r'_i, \sigma_{\mathcal{S},i})$ from the honest server, it

only continues when it receives the same $t_i$ that is contained in the sign-sim record.

The challenger then maintains incomplete records $(\mathsf{setup\text{-}sim}, sid, k, \perp_{h_p}, d_{\mathcal{S}}, d_{\mathcal{D}}, (N, e))$ and $(\mathsf{sign\text{-}sim}, sid, qid_i, \perp_{h'_p}, h_{m,i}, m_i, \{t_i, \perp_t\}, r_i, \{r'_i, \perp_{r'}\}, \{\sigma_{\mathsf{RSA},i}, \perp_\sigma\})$ where $t_i$ is only assigned when the honest server sends its signature share. This event might not occur though, e.g., because the signing request never arrived or the environment didn't gave the ok to proceed. Similarly, also the list $\mathcal{Q}$ only contains incomplete entries $(C', (\perp_{h'_p}, \perp_{h_m}, \{t_i, \perp_t\}), (sid, qid_i))$. We we will fully complete those tuples and finally assign values to $h_p$ and $h'_{p,i}$ as soon as the server gets corrupted.

When the server gets corrupted, and the device is still honest, $h_p$ and $t_i$ are chosen at random, whereas $h'_{p,i}$ is determined based on $c_i$. That is, if $c_i = \mathsf{pwdok}$ then $h'_{p,i} \leftarrow \mathcal{H}(qid_i, h_p)$ and $h'_{p,i} \xleftarrow{r} \{0,1\}^\tau$ otherwise. The rest of the simulation for this setting is then equivalent to the previous game, where we already started with a corrupt server. Thus, in the remainder of this game we will focus on the setting where the device gets corrupted first.

If the device gets corrupted (and the server is still honest), the adversary learns $k$ and $r_i$, and thus is now able to compute the hashes $h_p, h'_p$ and tag $t_i$ himself. However, it has not learned *our* choices for those values yet, since the server is still honest. And in fact, we will still not assign any values for $h_p, h'_p$ or $t_i$ (for the open signing requests).

As in the game above, we abort if we receive a random oracle query of the form $\mathcal{H}(k, \cdot)$ or $\mathcal{H}(\texttt{"MAC"}, \cdot, k, \cdot)$ where $k$ is the secret device key and the device is still honest. Again, as we have not used $k$ in any computation so far, such a query can only occur with negligible probability. Then, as soon as the device gets corrupted and the adversary learns $k$, we do no longer abort for those queries. The procedure to answer queries of the form $\mathcal{H}(\texttt{"MAC"}, qid_i, k, h_{m,i})$ is the same as in the game described above. Whereas for fresh queries of the form $\mathcal{H}(k, pwd)$ or $\mathcal{H}(qid_i, h_p^*)$ that might be an attempt to (re)-compute $h_p$, $h'_p$ we simply assign random values.

If the device gets corrupted while the server is still honest, we also change the way the server verifies whether the provided authentication information for a new signing request is correct. For each such signing request $(sid, qid_i, C'_i)$ coming from a corrupted (but initially honest) device, the honest server first decrypts $(h'_{p,i}, h_{m,i}, t_i) \leftarrow \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{decrypt}, C'_i, (sid, qid_i))$ using the simulator of the non-committing encryption scheme. Then, the challenger checks if the random oracle table $\mathcal{T}_{\mathcal{H}}$ contains a preimage $(qid_i, h_p)$ for $h'_{p,i}$ and a preimage $(k, pwd)$ for the retrieved $h_p$ and having the correct device key $k$ as prefix ($k$ is stored in setup-sim for $sid$). If such a preimage exists, we create a record $(\mathsf{pwdsign}, sid, qid_i, pwd'_i)$ with $pwd'_i \leftarrow pwd$ and verify the correctness of the password based on the pwdsign and pwdsetup records. The rest of the signing protocol is handled as with a device that was corrupt from the beginning.

When eventually also the server gets corrupted, the adversary will learn the key $esk$ to decrypt the communication towards the server and thus could now

check if we have chosen the "correct" $h_p, h'_p$ values before. Thus, this is the moment we finally determine those hashes. To this end, the challenger checks if it already responded to random oracle queries $\mathcal{H}(k, pwd)$ or $\mathcal{H}(qid_i, h^*_p)$ where $((k, pwd'_i), h^*_p) \in \mathcal{T}_{\mathcal{H}}$, for the passwords $pwd$ and $pwd'_i$ stored in the pwdsetup and pwdsign records. If such queries are found, the challenger sets the value for $h_p, h'_{p,i}$ to be the random oracle answers it had given earlier. If no queries of that form were made, it assigns random hash values. Similarly, the tags $t_i$ of incomplete signature requests are either chosen at random or by reusing the response for a query $\mathcal{H}(\texttt{"MAC"}, qid_i, k, h_{m,i})$ the adversary has made earlier. The challenger also updates its setup-sim and sign-sim records and the tuples in $\mathcal{Q}$ to contain the determined values for $h_p, h'_{p,i}$, and $t_i$ and finally derives and outputs the decryption key $esk \leftarrow \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{randomness}, \mathcal{Q})$.

Again, by the random choice of $k \xleftarrow{r} \{0,1\}^\tau$, the environment can distinguish this game hop only with negligible probability.

Game 8: In this game, we modify the signing procedure when done between an honest device and corrupt server. Roughly, the goal is to make the signature somewhat independent of $m_i$, in the sense that we do not use it in the protocol simulation or in the sign-sim record, but we do register it in a signature record at the end. The connection to the real message $m_i$ is only established when the adversary tries to verify the signature and therefore makes a random oracle query $(r_i, m_i)$.

To this end, when an honest device wants to sign a message $m_i$, we choose $r_i, h_{m,i}$ at random from $\{0,1\}^\tau$ and store all values except $m_i$ associated with the jointly computed signature $\sigma_i \leftarrow (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$, where $r'_i$ was provided by the server. That is, the challenger then has a record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \perp_m, t_i, r_i, r'_i, \sigma_{\mathsf{RSA},i})$ for the signing process and another one for the completed signature as $(\mathsf{signature}, sid, qid_i, m_i, \sigma_i)$.

If the device did not receive the server's contribution $(\sigma_{\mathcal{S},i}, r'_i)$, the challenger only has a record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \perp_m, t_i, r_i, \perp_{r'}, \perp_\sigma)$. However, it will fill in the missing $r'_i$ and the signature itself as soon as the device gets corrupted. More precisely, the challenger chooses a random $r'_i$ and computes $\sigma_{\mathsf{RSA},i} \leftarrow \mathcal{H}_{\mathsf{RSA}}(sid, qid_i, \mathcal{H}(r'_i, h_{m,i}))^{d_{\mathcal{S}}\, d_{\mathcal{D}}}$ using the knowledge of $d_{\mathcal{D}}$ *and* $d_{\mathcal{S}}$ as the setup was done by an honest device. It then updates the sign-sim record accordingly and also creates the corresponding signature record.

So far, we have made the signature independent of $m_i$, as we have signed a random $h_{m,i}$ value and did not program the random oracle to map $(r_i, m_i)$ to $h_{m,i}$ yet. The challenger then "fixes" this when the corresponding message $(r_i, m_i)$ is queried to $\mathcal{H}$ by the adversary using the sign-sim and signature records.

That is, whenever the challenger receives a random oracle query of the form $(r_i, m_i)$ it first checks whether it has a matching record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \perp_m, t_i, r_i, \perp_{r'}, \perp_\sigma)$ containing the same $qid_i, r_i$. If it has such a record but $r'_i = \sigma_{\mathsf{RSA},i} = \perp$, the challenger aborts. Note that this case only occurs if the server never provided its contribution and the device is still honest, which in turn means the adversary did not learn $r_i$ so far but (at most) the random

value $h_{m,i}$. Due to the choice of $r_i \xleftarrow{r} \{0,1\}^\tau$, a query $\mathcal{H}(r_i, m_i)$ for the same $r_i$ can only appear with negligible probability then.

If the sign-sim record was completed though (except of $m_i$), and a signature record $(\mathsf{signature}, sid, qid_i, m_i, \sigma_i)$ for the queried message $m_i$ and with $\sigma_i = (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$ exists, the challenger responds by programming $\mathcal{H}(r_i, m_i)$ to $h_{m,i}$ using the previously chosen $h_{m,i}$ value from the sign-sim record. When no matching signature record exists, the challenger simply sets the random oracle response to a random value.

Such a "just-in-time" programming of the random oracle is not possible if a record $((r_i, m_i), h^*) \in \mathcal{T}_\mathcal{H}$ with $h^* \neq h_{m,i}$ already exists, i.e., the adversary had "predicted" the same $r_i$ in a random oracle query before the honest device has randomly chosen $r_i$ from $\{0,1\}^\tau$ during a signing request. However, given that the adversary makes at most $q_h$ queries to the random oracle and the honest device participates in at most $q_s$ signing sessions, the adversary has an advantage of at most $\frac{q_h q_s}{2^\tau}$ to hit that event. Thus, the environment can distinguish that game hop with negligible probability only.

Game 9: We now do a similar change as in the previous game but for the setting where an honest device runs a signing protocol with an *honest* server. The difference to the procedure above is that the device here only draws a random $r_i$, while $h_{m,i}$ gets chosen at the moment when the honest server creates its signature share. (Recall that since Game 3 the honest device only sends a simulated ciphertext to the server and Step 2 of the signing protocol is done solely based on the internal record maintained by the challenger.)

Thus, after completing Step 1 of the signing protocol, the challenger only maintains a record $(\mathsf{sign\text{-}sim}, sid, qid_i, \bot_{h'_p}, \bot_{h_m}, \bot_m, \bot_t, r_i, \bot_{r'}, \bot_\sigma)$ (the way $h'_{p,i}$ and $t_i$ are computed was already changed in Game 7). When the server then gets the ok to proceed and wants to create its signature share, it chooses a random $h_{m,i}$ and does the rest according to the protocol. The challenger also updates its record to $(\mathsf{sign\text{-}sim}, sid, qid_i, \bot_{h'_p}, h_{m,i}, \bot_m, t_i, r_i, r'_i, \bot_\sigma)$ where $t_i$ is determined as described in Game 7. If the honest device now receives a message $(sid, qid_i, h_{m,i}, t_i, r'_i, \sigma_{\mathcal{S},i})$ from the honest server, it further completes the sign-sim record to $(\mathsf{sign\text{-}sim}, sid, qid_i, \bot_{h'_p}, h_{m,i}, \bot_m, t_i, r_i, r'_i, \sigma_{\mathsf{RSA},i})$ and also creates a signature record $(\mathsf{signature}, sid, qid_i, m_i, \sigma_i)$.

As in the game above, we use the completed sign-sim and signature records to consistently answer random oracle queries $(r_i, m_i)$, while we abort if such a query is made but only an incomplete signing record exists and the device is still honest.

Similar as in the game above, we then change that behavior when the device gets corrupted and also complete the records of interrupted or discontinued signing sessions. However, here we have to cope with the additional case that the server might still be honest when the device got corrupted. Thus, we let the completion and random-oracle handling depend on the status of the adaptive corruption.

In any case, if the device gets corrupted, we have to provide the $r_i$ values of all signing requests, including ongoing ones, to the adversary. That is, the adversary now knows the randomness that was allegedly used to compute the (possibly still unassigned) $h_{m,i}$ value. Thus, we complete the signing records of interrupted signing sessions such that we can do the "just-in-time" random oracle programming for queries $(r_i, m_i)$ as in the previous game. However, if the server is still honest we will complete only records of the form $(\mathsf{sign\text{-}sim}, sid, qid_i, \bot_{h'_p}, h_{m,i}, \bot_m, t_i, r_i, r'_i, \bot_\sigma)$, i.e., where the honest server had already sent its signature share. For those, the challenger computes the full signature and creates a record $(\mathsf{signature}, sid, qid_i, m_i, \sigma_i)$.

In difference to the previous game that handled the setting of an (initially) honest device and corrupt server, we here also respond to random oracle queries $(r_i, m_i)$ for which only a record $(\mathsf{sign\text{-}sim}, sid, qid_i, \bot_{h'_p}, \bot_{h_m}, \bot_m, \bot_t, r_i, \bot_{r'}, \bot_\sigma)$ exist. If such a query occurs, the challenger responds with a random $h_{m,i}$ and adds $((r_i, m_i), h_{m_i})$ to $\mathcal{T}_\mathcal{H}$. We will complete those rather empty sign-sim records at the moment when both, the device and server, are corrupt.

That is, as soon as the server gets corrupted (too), the challenger completes those signatures, but by signing a random $h'_{m,i}$ and choosing a random $r'_i$. That is, $h'_{m,i}$ is not a "proper" random oracle response yet. The challenger also updates its sign-sim record to include $h'_{m,i}, r'_i, \sigma_{\mathsf{RSA},i}$ and creates a full signature record. Thus, while $(\mathsf{signature}, sid, qid_i, m_i, \sigma_i)$ is complete now, the record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, \bot_{h_m}, \bot_m, t_i, r_i, r'_i, \sigma_{\mathsf{RSA},i})$ still misses the $h_{m,i}$ value. (The way $h'_{p,i}$ is chosen is described in Game 7.)

The challenger then determines the "correct" value for $h_{m,i}$ by going through all answered random oracle queries that have the form $((r_i, m_i), h_{m,i}) \in \mathcal{T}_\mathcal{H}$. For each such entry the challenger checks if a matching signature record $(\mathsf{signature}, sid, qid_i, m_i, \sigma_i)$ exists, i.e., the record contains the same $m_i$ and $\sigma_i = (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$ contains $r_i$. If that is the case, the challenger now fully completes the sign-sim record by including $h_{m,i}$ (taken from $\mathcal{T}_\mathcal{H}$) and $m_i$ (taken from signature). It also sets $\mathcal{H}(r'_i, h_{m,i}) \leftarrow h'_{m,i}$, i.e., it links $h_{m,i}$ to the random hash value $h'_{m,i}$ it has signed. If there is no such matching random oracle query $((r_i, m_i), h_{m,i}) \in \mathcal{T}_\mathcal{H}$, the challenger chooses a random $h_{m,i} \xleftarrow{r} \{0,1\}^\tau$, sets $\mathcal{H}(r'_i, h_{m,i}) \leftarrow h'_{m,i}$ and also updates its sign-sim record to contain $h_{m,i}$. Thus, all missing values $h_{m,i}$ get assigned as soon as the server gets corrupted. The challenger then uses those sign-sim records to complete all tuples $(C', (h'_{p,i}, h_{m,i}, t_i), (sid, qid_i)$ for $\mathcal{Q}$. Recall that the complete list of all ciphertext/plaintext pairs is needed to get $esk \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{randomness}, \mathcal{Q})$.

From now on, every new query $\mathcal{H}(r_i, m_i)$ is answered as in the previous game, that is, by checking if a corresponding sign-sim record exists, in which case the random oracle response is set to $h_{m,i}$ contained in sign-sim.

Again, such programming would fail if the adversary already queried $\mathcal{H}(r_i, \cdot)$ or $\mathcal{H}(r'_i, \cdot)$ and the challenger has subsequently chosen the same $r_i, r'_i$ values in a signing protocol.

However, all $r_i, r_i'$ are chosen at random from $\{0,1\}^\tau$, i.e., such an event can only occur with negligible probability. Thus, the environment can recognize this game hop only with negligible probability too.

GAME 10: In our final game we make the transition from letting the challenger run the "real" protocol (w.r.t. GAME 9) to letting him interact with the ideal functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$ and simulate all messages based solely on the information he can obtain from $\mathcal{F}_{\mathsf{Pass2Sign}}$. The description of this simulator is given in the following section.

### D.2 Simulator

We now complete the proof of Theorem 3 by describing how we construct a simulator $\mathsf{SIM}$ such that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$ that controls a certain subset of the parties, the view of the environment in the real world, when running the protocol (according to GAME 9 from Section D.1) with the adversary, is indistinguishable from its view in the ideal world where it interacts with the ideal functionality and the simulator (which corresponds to the final game from Section D.1).

We denote by "$\mathcal{D}$", "$\mathcal{S}$" the simulated honest party $\mathcal{D}$ and $\mathcal{S}$ respectively in the real world. The description of the simulator is given as follows: Section D.2.1 describes the setup procedure for the different combinations of honest and corrupt parties. Analogously, Section D.2.2 describes the signing process for those combinations. The simulation of the random oracle is then described in Section D.2.3 and the handling of adaptive corruptions is given in Section D.2.4. For simplicity, we refer to $\mathcal{F}_{\mathsf{Pass2Sign}}$ as $\mathcal{F}$ from now on.

#### D.2.1 Simulation of the Setup Protocol

When the server is initially honest, "$\mathcal{S}$" creates its public key of the non-committing encryption scheme as $epk \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{publickey}, 1^\tau)$ instead of using the real key generation.

*Setup – Step 1 (done by honest device "$\mathcal{D}$"):* The simulation starts when $\mathsf{SIM}$ receives a message $(\mathsf{SETUPREQ}, sid, )$ from $\mathcal{F}$, and then depends whether the account is created with an honest or corrupt server. In both cases, though, $k, (N,e), d_{\mathcal{S}}$ and $d_{\mathcal{D}}$ are generated according to the real protocol.

*Server is honest.* When "$\mathcal{D}$" is supposed to send an encryption of the secret key share and its authentication information, it sends a simulated ciphertext $C \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{encrypt}, |N|+\tau, (sid,(N,e)))$ instead. The simulator also internally stores the tuple $(C, (d_{\mathcal{S}}, \perp_{h_p}), (sid,(N,e)))$ in a list $\mathcal{Q}$. It will determine a value for $h_p$ as soon as the server gets corrupted. The simulator then creates an internal request record $(\mathsf{setup\text{-}req\text{-}sim}, sid, k, \perp_{h_p}, d_{\mathcal{S}}, d_{\mathcal{D}}, (N,e))$. The record will be transformed into an activated setup record when the request arrives at the server.

*Server is corrupt.* Here the password hash $h_p$ is chosen at random and "$\mathcal{D}$" sends the correct ciphertext $C$ and not a simulated one. The simulator then maintains a full setup record $(\mathsf{setup\text{-}sim}, sid, k, h_p, d_{\mathcal{S}}, d_{\mathcal{D}}, (N, e))$.

*Setup – Step 2 (done by honest server "$\mathcal{S}$"):* The simulation of an honest server "$\mathcal{S}$" starts when "$\mathcal{S}$" receives $(sid, (N, e), C)$ and "$\mathcal{S}$" does not have an account for $sid$ yet. If the tuple $(sid, (N, e), C)$ was created by an honest device, the simulation continues with the first case, otherwise with the second case.

*Request from honest device.* If "$\mathcal{S}$" receives the same simulated ciphertext $C$ that was sent by "$\mathcal{D}$", it does not decrypt the ciphertext but directly responds by sending $sid$. The simulator also stores an activated setup record as $(\mathsf{setup\text{-}sim}, sid, k, \perp_{h_p}, d_{\mathcal{S}}, d_{\mathcal{D}}, (N, e))$, using the information from $\mathsf{setup\text{-}req\text{-}sim}$.

*Request from corrupt device.* Here the server "decrypts" the ciphertext $C$ with label $(sid, (N, e))$ using the simulator of the non-committing encryption scheme as $(d_{\mathcal{S}}, h_p) \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{decrypt}, C, (sid, (N, e)))$ and stores the received information in an activated setup record as $(\mathsf{setup\text{-}sim}, sid, \perp_k, h_p, d_{\mathcal{S}}, \perp_{d_{\mathcal{D}}}, (N, e))$.

In a very special case the simulator might already have a different request record $(\mathsf{setup\text{-}req\text{-}sim}, sid, k', \perp_{h_p}, d'_{\mathcal{S}}, d'_{\mathcal{D}}, (N', e'))$ for the same $sid$. This can happen if the request was initiated by an honest device, but never reached the honest server. If the adversary then corrupts the device, it can "reuse" the same $sid$ from the honestly started setup request, but replace all other information. If the simulator notices such a replacement, it sends $(\mathsf{KEYGEN}, sid, 1, (N, e))$ to $\mathcal{F}$ which reflects the intrusion of the adversary and overwrites the initial password of the honest device in the ideal world by a dummy password "1".

If $\mathsf{SIM}$ does not have a $\mathsf{setup\text{-}req\text{-}sim}$ record for $sid$ yet, it sends $(\mathsf{SETUPREQ}, sid, 1)$ to $\mathcal{F}$ and subsequently inputs $(\mathsf{KEYGEN}, sid, \perp, (N, e))$ to $\mathcal{F}$.

Note that $\mathsf{SIM}$ uses "1" as the password of the corrupted device when creating, or overwriting the account in $\mathcal{F}$ and not the preimage of $h_p$, as such a preimage does not necessarily exist yet. For the further simulation this is sufficient though, as we only have to ensure that we invoke $\mathcal{F}$ either with the correct password (i.e., again with "1") or a wrong one (e.g., with "0").

*Setup – Step 3 (done by honest device "$\mathcal{D}$"):* When "$\mathcal{D}$" outputs $(\mathsf{SETUP}, sid, (N, e))$, the simulator registers the public key in the functionality by sending $(\mathsf{KEYGEN}, sid, \perp, (N, e))$ to $\mathcal{F}$, which will also deliver the message $(\mathsf{SETUP}, sid, (N, e))$ to the honest device in the ideal world.

### D.2.2 Simulation of the Signing Protocol

*Sign – Step 1 (done by honest device "$\mathcal{D}$"):* When $\mathsf{SIM}$ receives a message $(\mathsf{SIGNREQ}, sid, qid)$ from $\mathcal{F}$, it starts the simulation of "$\mathcal{D}$" which now has to

initiate a signing protocol in the real world, but without knowing $m$ or $pwd'$. Again, the simulation branches depending on whether "$\mathcal{D}$" interacts with an honest or corrupt server.

*Server is honest.* Here "$\mathcal{D}$", instead of sending the real ciphertext $C'$ to "$\mathcal{S}$", sends a simulated ciphertext $C' \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{encrypt}, 3\tau, (sid, qid))$ using the simulator of the non-committing encryption scheme. At this point, the simulator does not know the entire corresponding plaintext. Namely, it does not know $h_m, t$ yet and also does not have enough information to "correctly" determine $h'_p$. Thus, SIM stores the incomplete ciphertext/plaintext tuple $(C', (\bot_{h'_p}, \bot_{h_m}, \bot_t), (sid, qid))$ in $\mathcal{Q}$.

The device should also have created an internal signing record. Therefore, "$\mathcal{D}$" chooses a random $r \xleftarrow{r} \{0,1\}^\tau$ and stores it as $(\mathsf{sign}, sid, qid, r)$ and also initiates a signature request record $(\mathsf{sign\text{-}req\text{-}sim}, sid, qid, \bot_{h'_p}, \bot_{h_m}, \bot_m, \bot_t, r, \bot_{r'}, \bot_\sigma)$ that will be used for a consistent simulation. Similar as in setup, the request record will become a "real" sign record as soon as the request arrives at the honest server.

*Server is corrupt.* In this case, the simulator continues by sending $(\mathsf{DELIVER}, sid, qid, \bot, \bot)$ to $\mathcal{F}$ in order to learn whether the submitted password was correct. That is, when SIM then receives $(\mathsf{SIGNREQ}, sid, qid, status)$ from $\mathcal{F}$ with $status = \mathsf{pwdok}$ it retrieves its setup record $(\mathsf{setup\text{-}sim}, sid, k, h_p, d_\mathcal{S}, d_\mathcal{D}, (N, e))$ for $sid$ and computes $h'_p \leftarrow \mathcal{H}(qid, h_p)$. If SIM learns that the password did not match, i.e., $status = \mathsf{pwdwrong}$, the simulator chooses $h'_p \xleftarrow{r} \{0,1\}^\tau$ at random. Recall that SIM does not know the actual message that should be signed, but here the device has to send a correctly computed ciphertext $C' \xleftarrow{r} \mathsf{Enc}(epk, (h'_p, h_m, t), (sid, qid))$ to $\mathcal{S}$. Thus, in addition to $r$, "$\mathcal{D}$" also chooses random $h_m \xleftarrow{r} \{0,1\}^\tau$ and $t \xleftarrow{r} \{0,1\}^\tau$. All values $r, h_m, h'_p, t$ are kept in an internal record $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, \bot_m, t, r, \bot_{r'}, \bot_\sigma)$. The simulated device "$\mathcal{D}$" then sends the message $(sid, qid, C')$ to $\mathcal{S}$.

*Sign – Step 2 (done by honest server "$\mathcal{S}$"):* The simulation of an honest server starts when "$\mathcal{S}$" receives a message $(sid, qid, C')$, and then branches depending on whether the message was sent by an honest device or corrupt device. That is, even if the device got corrupted in the meantime, but the adversary has not replaced the initially sent message of "$\mathcal{D}$", the first case applies.

*Request from honest device.* If $(sid, qid, C')$ was sent by an honest device, "$\mathcal{S}$" does not decrypt $C'$ but directly sends $(\mathsf{DELIVER}, sid, qid, \bot, \bot)$ to $\mathcal{F}$, triggering the output to $\mathcal{S}$ in the ideal world. The simulator also reflects the arrived request by storing a record $(\mathsf{sign\text{-}sim}, sid, qid, \bot_{h'_p}, \bot_{h_m}, \bot_m, \bot_t, r, \bot_{r'}, \bot_\sigma)$ using the information form $\mathsf{sign\text{-}req\text{-}sim}$.

*Request from corrupt device.* When a request came from a corrupt device, we must further distinguish whether or not it replaces a pending sign request that was initiated from the device when it was still honest.

If it replaces another request, the simulator already maintains a $\mathsf{sign\text{-}req\text{-}sim}$ record for the same $qid$. The simulator then first decrypt $C'$ with the

help of the simulator of the non-committing encryption scheme and obtains $(h'_p, h_m, t) \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{decrypt}, C', (sid, qid))$. The simulator must now reflect the replacement of the sign request towards the ideal functionality too. That is, it must determine $pwd^*$ and $m^*$ which should replace the initial password and message. To determine the message, the simulator uses its maintained random-oracle table $\mathcal{T}_{\mathcal{H}}$ to look up the preimage $(r, m^*)$ for $h_m$. If no preimage of the form $(r, m^*)$ exists, SIM sets $m^* \leftarrow \perp$, which it will solely use towards the ideal functionality. Note, that all calls to $\mathcal{H}$ are also implicitly prefixed with the *sid* of the current session/account. That is, the adversary cannot reuse a dummy $h_m$ that was chosen at random in another *sid*-session by an honest device to obtain a valid signature on the unknown message.
The simulator proceeds similarly to obtain the adversaries password attempt: SIM first checks if the random oracle table $\mathcal{T}_{\mathcal{H}}$ contains a preimage $(qid, h_p)$ for $h'_p$ and a preimage $(k, pwd^*)$ for the retrieved $h_p$ and with the device key $k$ as prefix (which is stored in setup-sim for *sid*). If such a proper preimage was found, SIM uses the retrieved $pwd^*$ or sets $pwd^* \xleftarrow{r} \{0,1\}^\tau$ to a random value otherwise, which will mimic a signature request for a wrong password. Finally, it sends $(\mathsf{DELIVER}, sid, qid, pwd^*, m^*)$ to $\mathcal{F}$ which triggers the output of $(\mathsf{SIGNREQ}, sid, qid, status)$ to the honest server in the ideal world. The honest server also maintains a record $(\mathsf{sign}, sid, qid, h_m, t, c)$ for the signing request, where $c \leftarrow \mathsf{pwdwrong}$ if no proper $pwd^*$ was found in the random oracle, and $c \leftarrow \perp$ otherwise (as SIM doesn't know yet whether the password was correct or wrong)
For the case where the request was fully initiated from a corrupt device, the simulator uses the same strategy as above to decrypt $(h'_p, h_m, t)$ and determine the message $m$. However, the password attempt is derived differently as here the server maintains a complete setup record $(\mathsf{setup}, sid, h_p, d_{\mathcal{S}}, (N, e))$ for *sid*. Thus, "$\mathcal{S}$" can use the stored value $h_p$ to normally verify whether $\mathcal{H}(qid, h_p) = h'_p$. If that is the case, SIM sets $pwd' \leftarrow 1$ and $pwd' \leftarrow 0$ otherwise, where $pwd'$ denotes the password that the simulator will use towards the ideal functionality (recall that we set $pwd \leftarrow 1$ in $\mathcal{F}$ for an account generated by the simulator). The simulator then initiates a signing session in $\mathcal{F}$ by sending $(\mathsf{SIGNREQ}, sid, qid, pwd', m)$ followed by $(\mathsf{DELIVER}, sid, qid, \perp, \perp)$. Here, the honest server also maintains a full record $(\mathsf{sign}, sid, qid, h_m, t, c)$ for the signing request.
In both cases, the honest server also creates activated sign records now. For values $h_m$ where a proper preimage $(r, m)$ in $\mathcal{T}_{\mathcal{H}}$ existed, SIM stores a sign record $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, m, t, r, \perp_{r'}, \perp_\sigma)$. That is, here SIM knows the message $m$ that it is supposed to sign. If no preimage of the form $(r, m)$ existed, the simulator only creates a record $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, \perp_m, t, \perp_r, \perp_{r'}, \perp_\sigma)$.

*Sign – Step 3 (done by honest server "$\mathcal{S}$"):* When SIM receives a message $(\mathsf{PROCEED}, sid, qid)$ from $\mathcal{F}$ it knows that the password $pwd'$ provided by the device was correct and the honest server in the ideal world approves the signing request. Thus, "$\mathcal{S}$" acts accordingly and creates its signature contribution $\sigma_{\mathcal{S}}$.

Depending on whether "$\mathcal{S}$" has received the signing request from an honest or corrupt device, the simulator might not have chosen a value for $h_m, t$ yet, and thus the simulation again branches:

*Request from honest device.* Here, "$\mathcal{S}$" only received a dummy ciphertext $C'$ in the previous step, and the simulator has not assigned a value to $h_m, t$ or $h'_p$ yet. Thus, "$\mathcal{S}$" now chooses $h_m \xleftarrow{r} \{0,1\}^\tau$ and $r' \xleftarrow{r} \{0,1\}^\tau$ and uses the secret key $d_\mathcal{S}$ stored in the setup-sim record for $sid$ to compute $\sigma_\mathcal{S} \leftarrow \mathcal{H}_{\mathsf{RSA}}(sid, qid, \mathcal{H}(r', h_m))^{d_\mathcal{S}}$. The simulator also draws $t \xleftarrow{r} \{0,1\}^\tau$ and updates its sign-sim record to $(\mathsf{sign\text{-}sim}, sid, qid, \bot_{h'_p}, h_m, \bot_m, t, r, r', \bot_\sigma)$.
The newly created information is also included in the list $\mathcal{Q}$ by updating the tuple with label $(sid, qid)$ to contain the almost full plaintext $(\bot_{h_p}, h_m, t)$. Recall that $h'_{p,i}$ is only assigned as soon as the server gets corrupted, as described in Game 7. The server now also maintains a full record $(\mathsf{sign}, sid, qid, h_m, t, c)$ with $c \leftarrow \mathsf{pwdok}$ for the signing request. "$\mathcal{S}$" then sends $(sid, qid, h_m, t, r', \sigma_\mathcal{S})$ to "$\mathcal{D}$", if the device is still honest, or to $\mathcal{D}$ if it got corrupted in the meantime.

*Request from corrupt device.* Here, "$\mathcal{S}$" looks up its record $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, \{m, \bot_m\}, t, \{r, \bot_r\}, \bot_{r'}, \bot_\sigma)$ and normally computes its signature share $\sigma_\mathcal{S}$ for the stored $h_m$. In that process, "$\mathcal{S}$" chooses a random $r' \xleftarrow{r} \{0,1\}^\tau$ which is then included in the sign-sim record. Finally, "$\mathcal{S}$" sends $(sid, qid, h_m, t, r', \sigma_\mathcal{S})$ to $\mathcal{D}$. The honest server also updates its state record to $(\mathsf{sign}, sid, qid, h_m, t, c)$ setting $c \leftarrow \mathsf{pwdok}$.
Here, SIM now maintains a signature record of the form $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, \{m, \bot_m\}, t, \{r, \bot_r\}, r', \bot_\sigma)$ That is, SIM knows (for "well-formed" $h_m$) the message $m$ it has provided its signature share for, but not the created signature $\sigma$. In particular, also the ideal functionality $\mathcal{F}$ has not stored any signature for $m$ yet. However, we can provide the missing signature right on time when the environment tries to verify the signature, as described in Section D.2.3. Note that for non "well-formed" hashes $h_m$ we have initiated a signature request for dummy message $m = \bot$ towards the ideal functionality. However, such a request will never lead to a signature in $\mathcal{F}$ due to the collisision-resistance provided by the random oracle.

*Sign – Step 4 (done by honest device "$\mathcal{D}$"):* When "$\mathcal{D}$" receives a message $(sid, qid, h_m, t, r', \sigma_\mathcal{S})$, it only continues if the received value $t$ is the same as stored in $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, \bot_m, t, r, r', \bot_\sigma)$. It then completes the signature to $\sigma_{\mathsf{RSA}}$ using the locally stored $d_\mathcal{D}$ value. Eventually, "$\mathcal{D}$" ends with output $(\mathsf{SIGNATURE}, sid, qid, \sigma)$, upon which SIM sends $(\mathsf{SIGNATURE}, sid, qid, \sigma)$ to $\mathcal{F}$ and also stores $\sigma_{\mathsf{RSA}}$ in its internal sign-sim record. Thus, the simulator has then "blindly" generated a signature in $\mathcal{F}$, i.e., SIM has not learned the signed message $m$ yet, while the ideal functionality now contains a record $(\mathsf{signature}, (N, e), m, \sigma, \mathsf{true})$.

To summarize, depending on the interference of the adversary and the input of the environment we end this simulation for an *honest* device in one of the following states:

**Signature completed:** The signature process ended correctly and $\mathcal{F}$ contains a valid signature record including the message $m$ (which is unknown to the simulator) and the completed signature $\sigma$. The simulator created a signature record $(\mathsf{sign\text{-}sim}, sid, qid, \{h'_p, \perp_{h'_p}\}, h_m, t, \perp_m, r, r', \sigma_{\mathsf{RSA}})$, that contains $h_m$ and the created RSA signature $\sigma_{\mathsf{RSA}}$, and where $\perp_m$ stands for the unknown message $m$ that the simulator had signed. However, the simulator can use the fact that the ideal functionality contains a completed signature record *including* the message $m$ to ensure consistency when a random oracle query for the message $(r, m)$ is made. Furthermore, if done with an honest server "$\mathcal{S}$", "$\mathcal{S}$" also created a signing record $(\mathsf{sign}, sid, qid, h_m, t, c)$ where $c \leftarrow \mathsf{pwdok}$.

**Proceed, but no completed signature:** We ended in the second state when the environment gave the ok to proceed, but the adversary in the real world has interrupted the final message, such the device never received $(sid, qid, h_m, t, r', \sigma_{\mathcal{S}})$ from "$\mathcal{S}$". Consequently, we could not create the signature record in $\mathcal{F}$ yet, that we later need to ensure consistency with the random oracle. Also the simulator only maintains a record $(\mathsf{sign\text{-}sim}, sid, qid, \{h'_p, \perp_{h'_p}\}, h_m, t, \perp_m, r, r', \perp_\sigma)$. (Note that $r', t$ and $h_m$ were added to the record already when "$\mathcal{S}$" send its message.) However, we will finalize the computation of $\sigma$ as soon as the device gets corrupted using the already selected $r'$ and create the corresponding ideal world record by sending $(\mathsf{SIGNATURE}, sid, qid, \sigma)$ to $\mathcal{F}$. If an honest server was involved, it also stores a full record $(\mathsf{sign}, sid, qid, h_m, t, c)$ with $c \leftarrow \mathsf{pwdok}$.

**No proceed:** The third state occurs when the sign request arrived at the server, but the either the environment didn't gave the ok to proceed. Thus, we have neither created a signature $\sigma_{\mathcal{S}}$, nor allocated $r', t$ or $h_m$ yet. The record of SIM therefore looks as follows $(\mathsf{sign\text{-}sim}, sid, qid, \{h'_p, \perp_{h'_p}\}, \perp_{h_m}, \perp_m, \perp_t, r, \perp_{r'}, \perp_\sigma)$. An honest server is then also supposed to have created an intermediate record $(\mathsf{sign}, sid, qid, h_m, t, c)$ where $c$ indicates whether the password matched or not. However, in our simulation $h_m$ and $c$ are not known yet, and therefore "$\mathcal{S}$" only has the incomplete record $(\mathsf{sign}, sid, qid, \perp_{h_m}, \perp_t, \perp_c)$. The simulator will complete such records when the server gets corrupted.

**Signing request never arrived:** The last state occurs when the adversary already intercepted the signature request from "$\mathcal{D}$". Here, SIM only holds the following record $(\mathsf{sign\text{-}req\text{-}sim}, sid, qid, \{h'_p, \perp_{h'_p}\}, \perp_{h_m}, \perp_m, \perp_t, r, \perp_{r'}, \perp_\sigma)$.

### D.2.3 Verification and Random Oracle Simulation

In the simulation so far, SIM has sometimes "blindly" signed messages for a user, or signed messages which were known to the simulator, but where SIM did not learn the resulting signature. However, we can use the verification interface and the fact that SIM is in charge of answering the random oracle queries, to learn the missing values and ensure consistency with the values maintained by the ideal functionality $\mathcal{F}$.

First we show how SIM can learn the missing message for records of type $(\mathsf{sign\text{-}sim}, sid, qid, \{h'_p, \perp_{h'_p}\}, h_m, \perp_m, t, r, r', \sigma_{\mathsf{RSA}})$ or $(\mathsf{sign\text{-}req\text{-}sim}, sid, qid, \{h'_p, \perp_{h'_p}\}, h_m, \perp_m, t, r, r', \sigma_{\mathsf{RSA}})$. Our simulator uses the procedure as described in Game 8 and Game 9, with the modification that whenever the challenger would look up if a record $(\mathsf{signature}, sid, qid, m, \sigma)$ exists, SIM sends $(\mathsf{VERIFY}, sid, m, \sigma, (N, e))$ to $\mathcal{F}$, where $(N, e)$ is taken from the setup record for $sid$. When $\mathcal{F}$ asks SIM to verify a signature $\sigma$, the simulator returns $(\mathsf{VERIFIED}, sid, m, \sigma, (N, e), \mathsf{false})$ and waits for a message $(\mathsf{VERIFIED}, sid, m, \sigma, (N, e), f)$. If $f = \mathsf{true}$, SIM behaves as if the challenger would have found a matching record $(\mathsf{signature}, sid, qid, m, \sigma)$. Thereby we ensure that whenever a random oracle query $(r, m)$ is made where $r$ was used in a signing session, we can detect a query containing the "real" $m$ that the simulator might have blindly signed, and react consistently.

The second type of incomplete records were created when only the server was honest. In that case the simulator could "extract" the message, but did not learn the signature the corrupt device had completed. Thus, the simulator maintains a record $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, m, t, r, r', \perp_\sigma)$, and more importantly, the ideal functionality did not receive the signature either. Whenever SIM then receives a message $(\mathsf{VERIFY}, sid, m, \sigma, (N, e), \mathcal{P})$ from $\mathcal{F}$, it checks whether $\sigma$ is valid signature on $m$ using the normal verification algorithm. If verification fails, SIM responds with $(\mathsf{VERIFIED}, sid, m, \sigma, (N, e), \mathsf{false})$ to $\mathcal{F}$. When the verification succeeds, SIM parses $\sigma$ as $(\sigma_{\mathsf{RSA}}, qid, r, r')$ and checks if it has a $\mathsf{setup\text{-}sim}$ record for $sid$ with the same public key $(N, e)$ and also a record $(\mathsf{sign\text{-}sim}, sid, qid, h'_p, h_m, m, t, r, r', \perp_\sigma)$ for the same $sid, qid, m, r, r'$. If such matching records are found, i.e., the signature belongs to a previous signing request, SIM sends $(\mathsf{SIGNATURE}, sid, qid, \sigma)$ to $\mathcal{F}$, ensuring that the signature is now registered by the ideal functionality as well, followed by a call $(\mathsf{VERIFIED}, sid, m, \sigma, (N, e), \mathsf{true})$. When the verification in the real world succeeded but no matching record was found, SIM aborts, as justified in Game 5 and Game 6.

The simulator also takes special care of queries that contain the device secret key $k$ as described in Game 8 and Game 9. Whenever, in those games a record pwdsign or pwdsetup is used, SIM instead invokes the PWDGUESS interface of $\mathcal{F}$. We recall the procedure in more detail in the description of full corruption below.

### D.2.4 Adaptive Corruption

*Device Corruption.* When the environment decides to corrupt a previously honest device, "$\mathcal{D}$" receives the message $(\mathsf{corrupt}, sid)$ from the environment. This is mimiced by sending $(\mathsf{CORRUPT}, sid, \mathcal{D}, \emptyset)$ to $\mathcal{F}$ in the ideal world. The impact of that corruption depends on whether it happened after the setup was completed, or before, and whether or not the server is (still) honest.

**During setup:** If the device gets corrupted during setup, i.e., "$\mathcal{D}$" already had sent $(sid, (N, e), C)$ to the server but never completed the setup, it must

provide its setup record $(\mathsf{setup\text{-}temp}, sid, k, d_{\mathcal{D}}, (N, e))$ to the adversary. All values were correctly generated in the setup and thus, "$\mathcal{D}$" simply gives the record $(\mathsf{setup\text{-}temp}, sid, k, d_{\mathcal{D}}, (N, e))$ to $\mathcal{A}$. If the setup was done with an honest server which already replied with $(sid)$, but the adversary never let the message arrive at "$\mathcal{D}$", the simulator now also sends $(\mathsf{SETUP}, sid, (N, e))$ to $\mathcal{F}$. This ensures that the account will be activated in the ideal functionality as well.

**After setup:** When the device gets corrupted after the setup was done, the adversary expects to get the setup record $(\mathsf{setup}, sid, k, d_{\mathcal{D}}, (N, e))$ as well as all records $\{(\mathsf{sign}, qid_i, r_i)\}$ that result from signing requests.

Before the simulator outputs those records, it ensures that incomplete signature requests that were initiated by the device are now completed towards the ideal functionality. This is crucial to ensure consistency with a potential random oracle query $(r_i, m_i)$ as described in Section D.2.3. The type of discontinued signing request we can complete now depends on whether the server is (still) honest or not.

- *Server honest:* If the server is still honest, the simulator can only complete signing requests, where the honest server (in the ideal world) already gave the ok to proceed. That is, for those requests, where in the simulation "$\mathcal{S}$" had already send its contribution $(r'_i, \sigma_{\mathcal{S},i})$ to "$\mathcal{D}$" but the adversary intercepted the share. For each such intercepted signing session the simulator maintains a record $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_p, h_{m,i}, \perp_m, t_i, r_i, r'_i, \perp_\sigma)$ and now computes $(\sigma_{\mathsf{RSA},i}) \leftarrow \mathcal{H}_{\mathsf{RSA}}(sid, qid_i, \mathcal{H}(r'_i, h_{m,i}))^{d_{\mathcal{S}}\, d_{\mathcal{D}}}$ using the RSA secret key shares $d_{\mathcal{S}}, d_{\mathcal{D}}$ the honest device had generated in setup. It then sets $\sigma_i \leftarrow (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$, updates its record $\mathsf{sign\text{-}sim}$ to contain $\sigma_{\mathsf{RSA},i}$ and, most importantly, sends $(\mathsf{SIGNATURE}, sid, qid_i, \sigma_i)$ to $\mathcal{F}$.

  From now on the simulator must also answer to random oracle queries $(r_i, m_i)$ where $r_i$ belongs to a signing session where the honest server had never sent its signature share. In those cases the adversary has not learned $h_{m,i}$ though, and in fact, those are not even chosen by the simulator yet. Thus, $\mathsf{SIM}$ will respond with a random value $h_{m,i}$ for each such query. The simulator then has to ensure consistency for those queries as soon as the server gets corrupted as well.

- *Server corrupt:* If the server was initially honest, the simulator first completes the records as in the case of the honest server described above. However, given that here the server is corrupt, the simulator additionally creates signatures in $\mathcal{F}$ for those sessions, where i) either an initially honest server never got the approval to continue or ii) a corrupt server did not provide a (valid) signature share. For that type of discontinued signing requests, the simulator maintains records $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \perp_m, t_i, r_i, \perp_{r'}, \perp_\sigma)$. In case those records stem from a setting where the server was initially honest, but got then corrupted, $h'_{p,i}, h_{m,i}, t_i$ got assigned at the moment of the corruption (see description of server corruption).

We now complete each such sign-sim record by choosing $r'_i \xleftarrow{r} \{0,1\}^\tau$ and computing $\sigma_{\mathsf{RSA},i} \leftarrow \mathcal{H}_{\mathsf{RSA}}(sid, qid_i, \mathcal{H}(r'_i, h_{m,i}))^{d_{\mathcal{S}}\, d_{\mathcal{D}}}$, again using the knowledge of $d_{\mathcal{S}}, d_{\mathcal{D}}$, generated by the honest device. SIM then sends $(\mathsf{PROCEED}, sid, qid_i)$ to $\mathcal{F}$, followed by the message $(\mathsf{SIGNATURE}, sid, qid_i, \sigma_i)$ where $\sigma_i \leftarrow (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$.

In both cases the simulator has now created full signature records in $\mathcal{F}$ that maps the still unknown message $m_i$ to the "blindly" created signature $\sigma_i$. This will allow the simulator to determine $m_i$ whenever a query $(r_i, m_i)$ is made to the random oracle from now on (which is exactly the crucial turning point, as the adversary will learn all the $r_i$ values of the incomplete signing records), using the procedure described in Section D.2.3.

*Server Corruption.* When the honest server "$\mathcal{S}$" receives the message $(\mathsf{corrupt}, sid)$ from the adversary, the adversary then expects to learn the secret key of the non-committing encryption scheme $esk$, the setup information $(\mathsf{setup}, sid, h_p, d_{\mathcal{S}}, (N, e))$ and records $\{(\mathsf{sign}, sid, qid_i, h_{m,i}, t_i, c_i)\}$ for all signing sessions.

However, $h_p$ might still be unassigned and the signing records will currently have the form $(\mathsf{sign}, sid, qid_i, \perp_{h_m}, \perp_t, \perp_c)$ though, whenever they were created in a signing protocol with an honest device and the server did not continue the protocol. However, we can assemble the correct records now using the list $\mathcal{L}$ that SIM will obtain from $\mathcal{F}$ and the internal signing records maintained by SIM. How the simulation proceeds again depends on whether the device is still honest or not.

- *Device honest:* If the device is still honest, the simulator sends $(\mathsf{CORRUPT}, sid, \mathcal{S}, \emptyset)$ to $\mathcal{F}$ receiving $(\mathsf{CORRUPT}, sid, \mathcal{S}, \mathcal{L})$. Then, for every record $(\mathsf{sign}, sid, qid_i, \perp_{h_m}, \perp_t, \perp_c)$ stored by "$\mathcal{S}$", the simulator takes $c_i$ from the tuple $(qid_i, c_i) \in \mathcal{L}$ and includes it in the sign record. Determining the missing values for $h_p$, $h'_{p,i}, h_{m,i}$ and $t_i$ is also rather simple here, as the adversary hasn't learned the device key $k$ and randomness $r_i$ yet that was used to "blind" the message $m_i$ in $h_{m,i}$. Thus, SIM also had not to react to random oracle queries of the form $(r_i, m_i)$ yet. Thus, for each incomplete sign-sim record the simulator simply chooses random values $h_{m,i} \xleftarrow{r} \{0,1\}^\tau$, $t_i \xleftarrow{r} \{0,1\}^\tau$. The password hash $h_p$ is chosen at random now and whenever $c_i = \mathsf{pwdok}$ it also sets the password attempt of that session to $h'_{p,i} \leftarrow \mathcal{H}(qid, h_p)$ and to $h'_{p,i} \leftarrow \{0,1\}^\tau$ otherwise. The created values of $h_p$, $h'_{p,i}, h_{m,i}$ are then added to the setup-sim and sign-sim records.
- *Device corrupted during setup:* When the adversary corrupted the initially honest device during setup it has learned the device secret $k$. Thus, the adversary would already have been able to make a random oracle query to compute the password hash himself that is supposedly encrypted in $C$. The simulator has to figure out whether he already committed to $h_p$ before outputting the secret decryption key $esk$.

  To do so, SIM sends $(\mathsf{CORRUPT}, sid, \mathcal{S}, \emptyset)$ to $\mathcal{F}$ receiving $(\mathsf{CORRUPT}, sid, \mathcal{S}, \mathcal{L})$ which will also enable the password guess interface.

The simulator then goes through previously answered random oracles queries that had the form $\mathcal{H}(k, pwd_j)$, and for each sends $(\mathsf{PWDGUESS}, sid, \bot, pwd_j)$ to $\mathcal{F}$ to verify whether $pwd$ was the actual password of the initially honest device. If $\mathcal{F}$ responds with $(\mathsf{PWDGUESS}, sid, \bot, c_j)$ with $c_j = \mathsf{pwdok}$ , it sets $h_p$ to be the random oracle answer it had randomly assigned for that query. If no such matching query is found, $h_p$ is chosen at random. To determine the decryption key, simulator invokes $esk \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{keyleak}, \mathcal{Q})$ with $\mathcal{Q} \leftarrow (C, (d_{\mathcal{S}}, h_p), (sid, (N, e))$ using $(N, e), d_{\mathcal{S}}$ from the $\mathsf{setup\text{-}req\text{-}sim}$ record. Finally, "$\mathcal{S}$" outputs $esk$ as well as all sign records. Note that here all sign records where already completed during the protocol run as all requests originated from a corrupt device, i.e., nothing has to be simulated here.

– *Device corrupted after setup:* When the device is corrupted somewhen after setup, we have to take special care of all signing request that were initiated when the device was still honest. Here the adversary when corrupting the device has not only learned the device secret $k$ but also all records $\{(\mathsf{sign}, qid_i, r_i)\}$ of signing requests from "$\mathcal{D}$". Thus, the adversary could have made random oracle queries targeted to compute the message hash himself. More precisely, the adversary would have been able to query $(r_i, m_i)$ to the random oracle where $m_i$ is the message for which the environment triggered the signing request. As the adversary will now also learn the decryption key $esk$, $\mathsf{SIM}$ has to make sure that all ciphertexts $C'_i$ sent by "$\mathcal{D}$" in the signing requests now open to the correct plaintext. That is, it must hold that $(h'_{p,i}, h_{m,i}, t_i) \leftarrow \mathsf{Dec}(esk, C'_i, (sid, qid_i))$ and $h_{m,i} \leftarrow \mathcal{H}(r_i, m_i)$.
This requires a more careful simulation using the functionality to determine the blindly signed messages. The simulator now completes the signature record in $\mathcal{F}$ and uses that record to determine whether it already answered to a random oracle query $(r_i, m_i)$. To this end, $\mathsf{SIM}$ first retrieves each $qid_i$ for which a request record $(\mathsf{sign\text{-}req\text{-}sim}, sid, qid_i, \bot_{h'_p}, \bot_{h_m}, \bot_m, \bot_t, r_i, \bot_{r'}, \bot_\sigma)$ exist and there is either a matching sign record of the form $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, *, *, *, *, *)$ or $(\mathsf{sign\text{-}sim}, sid, qid_i, \bot_{h'_p}, \bot_{h_m}, *, *, *, \bot_{r'}, \bot_\sigma)$. For each such $qid_i$, $\mathsf{SIM}$ chooses $r'_i \xleftarrow{r} \{0,1\}^\tau$, a *random* $h'_{m,i} \xleftarrow{r} \{0,1\}^\tau$ and computes the full signature $\sigma_{\mathsf{RSA},i} \leftarrow \mathcal{H}_{\mathsf{RSA}}(sid, qid_i, h'_{m,i})^{d_{\mathcal{S}}\, d_{\mathcal{D}}}$, again using the knowledge of $d_{\mathcal{S}}, d_{\mathcal{D}}$, generated by the initially honest device. $\mathsf{SIM}$ then updates it (empty) $\mathsf{sign\text{-}sim}$ record to contain $\sigma_{\mathsf{RSA},i}$ and $r'_i$. The simulator also adds $\sigma_i$ with $\sigma_i \leftarrow (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$ to a signature list $\Sigma$. If the list is complete, $\mathsf{SIM}$ finally sends $(\mathsf{CORRUPT}, sid, \mathcal{P}, \Sigma)$ to $\mathcal{F}$ which will generate full signature records in $\mathcal{F}$ for all blindly signed message $m_i$ that were incomplete so far. The input also enables the $\mathsf{PWDGUESS}$ interface, which will be crucial for the rest of the simulation.
We first leverage the fact that now all blindly signed messages are completed within $\mathcal{F}$ to ensure a concistent decryption for the dummy ciphertexts $C'_i$ of the sign protocols. The following procedure is done to either complete *request* records or sign records. In the former case, the initial honest sign request was

replaced after device corruption, whereas in the latter one the sign request was received by the server but never completed.

Now, SIM goes through all random oracle queries of the form $((r_i, m_i), h_{m,i}) \in \mathcal{T}_{\mathcal{H}}$ for which a sign-req-sim record for $r_i$ exist and sends $(\mathsf{VERIFY}, sid, m_i, \sigma_i, (N, e), \mathsf{SIM})$ to $\mathcal{F}$, where again $\sigma_i \leftarrow (\sigma_{\mathsf{RSA},i}, qid_i, r_i, r'_i)$ and $(N, e)$ is taken from the setup record for the $sid$ specified in sign-sim. The ideal functionality will then send its ping $(\mathsf{VERIFY}, sid, m_i, \sigma_i, (N, e), \mathsf{SIM})$ to SIM, upon which it responds with $(\mathsf{VERIFIED}, sid, m_i, \sigma_i, (N, e), \mathsf{false})$. In case $m_i$ is indeed the blindly signed message of signing request $qid_i$, the ideal functionality will respond with $(\mathsf{VERIFIED}, sid, m_i, \sigma_i, (N, e), \mathsf{true})$. Whenever that happens, SIM now knows that it had signed a message $m_i$ and also assigned already a hash value $h_{m,i} \leftarrow \mathcal{H}(r_i, m_i)$ for $m_i$. Thus, SIM includes that hash value $h_{m,i}$ in its internal record sign-sim or sign-req-sim. It also "links" the signed $h'_{m,i}$ value to $h_{m,i}$ by setting $\mathcal{H}(r'_i, h_{m,i}) \leftarrow h'_{m,i}$. If for a signing query $qid_i$ no such matching random oracle query $(r_i, m_i)$ was found, SIM draws a random hash $h_{m,i} \xleftarrow{r} \{0,1\}^{\tau}$ and updates its internal sign-sim or sign-req-sim record accordingly. In addition, it adds valid tags $t_i \leftarrow \mathcal{H}(\texttt{"MAC"}, qid, k, h_{m,i})$ to the records using $k$ from the corresponding setup-sim record.

We have to take similar care of the password hashes, as the adversary already knows $k$ and would have been able to compute the password hashes himself. Thus we use the procedure from Game 7 to determine $h_p$ and all $h'_{p,i}$ such that they are consistent with the adversaries view. However, in difference to Game 7 we don't have internal records for the password $pwd$ and all password attempts $pwd'_i$. Instead, SIM uses the PWDGUESS interface of $\mathcal{F}$, which is available now since both parties are corrupt. Thus, for all previously answered random oracles queries that had the form $\mathcal{H}(k, pwd)$, SIM sends $(\mathsf{PWDGUESS}, sid, \bot, pwd)$ to $\mathcal{F}$ to verify whether $pwd$ was the actual password of the initially honest device. If such a query is found, it sets $h_p$ to be the random oracle answer it had randomly assigned for that query. Likewise, for all queries $\mathcal{H}(qid_i, h_p^*)$ where $((k, pwd'_i), h_p^*) \in \mathcal{T}_{\mathcal{H}}$, SIM send $(\mathsf{PWDGUESS}, sid, qid_i, pwd'_i)$ to $\mathcal{F}$ and reuses its previous random oracle answers for each $h'_{p,i}$ it had already created.

Note that all internal records will now have the form $(\mathsf{sign\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \bot_m, t_i, r_i, r'_i, \sigma_{\mathsf{RSA},i})$ and a corresponding full signature record in $\mathcal{F}$ was created. That is, for all signature request that were ever started by the honest device, the simulator can now use $\mathcal{F}$ to ensure consistency with random oracle queries $(r_i, m_i)$ of still unknown messages $m_i$ (as described in Section D.2.3).

In the first and third cases, SIM now uses its sign-sim records to assemble the complete signing records $\{(\mathsf{sign}, sid, qid_i, h_{m,i}, t_i, c_i)\}$ the server is supposed to give to $\mathcal{A}$. Further, the simulator adds the newly obtained values to the list $\mathcal{Q}$ to contain the full plaintext tuples $(h'_{p,i}, h_{m,i}, t_i)$ for every simulated ciphertext $C'_i$. It also adds such tuples for all replaces signature request, i.e., where the simulator just completed the sign-req-sim records. Similarly, it adds $C, (d_{\mathcal{S}}, h_p), (sid, (N, e)$ to $\mathcal{Q}$ to ensure consistency for the dummy ciphertext of the setup protocol.

Finally, the simulator invokes $esk \xleftarrow{r} \mathsf{SIM}_{\mathsf{NCE}}(\mathsf{keyleak}, \mathcal{Q})$ to learn the secret key of the encryption scheme. "$\mathcal{S}$" then outputs $esk, (\mathsf{setup}, sid, h_p, d_{\mathcal{S}}, (N, e))$ and the completed records $\{(\mathsf{sign}, sid, qid_i, h_{m,i}, c_i)\}$ of all signing sessions.

*Simulating Offline Attacks after Full Corruption.* When the environment has fully corrupted both parties, the adversary in the real world learned $k, h_p, h'_{p,1}, \ldots, h'_{p,q}$ where $q$ denotes the number of signing requests initiated by the device. Thus, it can run offline attacks against the password hashes, trying to determine the underlying password. In fact, as the passwords of honest devices were provided by the environment, they might even be known to the adversary. We use $\mathcal{F}'s$ PWDGUESS interface that is available for fully-corrupted instances from now on for each random oracle query that looks like an attempt of the adversary to verify the password against a learned hash value.

That is, for each query $\mathcal{H}(k, pwd^*)$ where $k$ appears in a record $(\mathsf{setup\text{-}req\text{-}sim}, sid, k, h_p, d_{\mathcal{S}}, d_{\mathcal{D}}, (N, e))$, SIM sends $(\mathsf{PWDGUESS}, sid, \bot, pwd^*)$ to $\mathcal{F}$. When it receives $(\mathsf{PWDGUESS}, sid, qid, c)$ with $c = \mathsf{pwdok}$ from $\mathcal{F}$, the simulator sets $\mathcal{H}(k, pwd^*) \leftarrow h_p$ where $h_p$ is taken from the setup-sim record and to a random value when $c = \mathsf{pwdwrong}$.

Likewise, for every query $\mathcal{H}(qid_i, h_p^*)$ where $h_p^* = \mathcal{H}(k, pwd^*)$, i.e., $h_p^*$ is the result of a previous random oracle query $(k, pwd^*)$ and records $(\mathsf{setup\text{-}sim}, sid, k, h_p, d_{\mathcal{S}}, d_{\mathcal{D}}, (N, e))$ for $k$ and $(\mathsf{sign\text{-}req\text{-}sim}, sid, qid_i, h'_{p,i}, h_{m,i}, \{m_i/\bot_m\}, t_i, r_i, r'_i, \{\sigma_{\mathsf{RSA},i}, \bot_\sigma\})$ for $qid_i$ exist, SIM sends $(\mathsf{PWDGUESS}, sid, qid_i, pwd^*)$ to $\mathcal{F}$. When $\mathcal{F}$ responds with the message $(\mathsf{PWDGUESS}, sid, qid_i, c)$ where $c = \mathsf{pwdok}$, SIM sets $\mathcal{H}(qid_i, h_p^*) \leftarrow h'_{p,i}$ where $h'_{p,i}$ is taken from the sign-sim record and assigns a random response otherwise.

Thus, we have shown how to construct a simulator that provides a view that is indistinguishable to the one described in Game 10, which concludes our proof.

□

## E Experimental Results

We have implemented our Pass2Sign scheme (i.e., the one with message blindness) to have performance figures demonstrating its practicality. Summarized, setup with a decent security parameter (4,096 Bit Moduli for both the non-committing encryption and the signatures) takes roughly 20 seconds, while for each signature generation our non-optimized implementation takes far less than two seconds. In a real-life deployment, this performance is more than sufficient for most use-cases.

*Concrete Setting.* We measured the setup and sign protocol with three different RSA-moduli sizes, 1,024, 2,048 and 4,096 Bit to account for different security requirements. The key size is used for both the signing key and the RSA trapdoor permutation in the non-committing encryption scheme. To instantiate the random oracles $\mathcal{K}, \mathcal{G}$, and $\mathcal{H}$ we use SHA-512 and prefix each call accordingly.

The instantiation of the full-domain hash $\mathcal{H}_{\mathsf{RSA}}$ is based on the construction given in [6], and uses rejection sampling to uniformly map into $\mathbb{Z}_N^*$.

Our implementation uses Java 8. The server was run on a Intel i7-3740QM with 2.7GHz and 16GB RAM, while the device was a Nexus 10 with 1.7GHz, 2GB RAM and Android 5.1.1. We did not implement any optimization such as multi-threading, RSA-CRT, connection pooling, or keep-alive of connections. The communication partners send messages using standard TCP-Sockets, and open a new connection for each new sub-protocol. This time is included in our measurements. $\mathcal{F}_{\mathsf{Auth}}$ has been implemented using digital signatures with pre-shared keys. Calls to $\mathcal{F}_{\mathsf{CA}}$ are included in our measurements. However, the measurements do not contain network round-trip times (typically between 20ms and 400ms), as these clearly depend on the current locations of the server and the user, and only add an additional constant to our bare run-time measurements. For example, assuming a round-trip time of 100ms, one can roughly add these 100ms to the combined run-time.

As the timings should focus on our protocol, they do not include the generation of the setup parameters, such as the keys for the non-comitting encryption scheme, or the generation of the session and query identifiers *sid*, *qid*.

For all the following confidence intervals and tables 100 runs were measured.

*Setup Protocol.* For measuring the setup protocol, we fixed the password that is an input to our protocol to a fixed string. The runtime box-plots are in Figure 11 for the device and Figure 10 for the server. The corresponding percentiles are depicted in Table 2 and Table 3.

| | 1,024Bit | 2,048Bit | 4,096Bit |
|---|---|---|---|
| Min.: | 180.82 | 398.27 | 8′238.22 |
| 25th Percentile: | 443.77 | 2′063.02 | 10′760.03 |
| Median: | 648.11 | 3′335.34 | 14′343.46 |
| 75th Percentile: | 1′130.72 | 4′700.18 | 20′913.41 |
| 90th Percentile: | 1′527.36 | 6′629.75 | 37′721.16 |
| 95th Percentile: | 2′272.43 | 7′876.96 | 40′008.34 |
| Max.: | 3′927.65 | 14′649.61 | 57′822.42 |
| Average: | 855.58 | 3′646.27 | 16′202.58 |

**Table 2.** Percentiles for setup in ms for the device.

As it can be seen, the time for the setup on the device varies a lot, which results from the randomized RSA-key generation algorithm. Compared with the key generation on smart cards, our protocol (including communication time) is still significantly faster though. For a 2,048 Bit modulus, even rather powerful smart cards take more than one minute for an RSA key pair generation[3], whereas

[3] http://www.pronew.com.tw/download/doc/400 Smart Card 080907.pdf

| | 1,024Bit | 2,048Bit | 4,096Bit |
|---|---|---|---|
| Min.: | 11.80 | 55.50 | 363.10 |
| 25th Percentile: | 13.22 | 61.07 | 376.53 |
| Median: | 14.33 | 63.96 | 388.10 |
| 75th Percentile: | 16.14 | 68.22 | 396.01 |
| 90th Percentile: | 17.23 | 74.40 | 428.47 |
| 95th Percentile: | 19.85 | 83.44 | 444.75 |
| Max.: | 48.92 | 99.93 | 478.49 |
| Average: | 15.20 | 65.69 | 393.27 |

**Table 3.** Percentiles for setup in ms for the server.

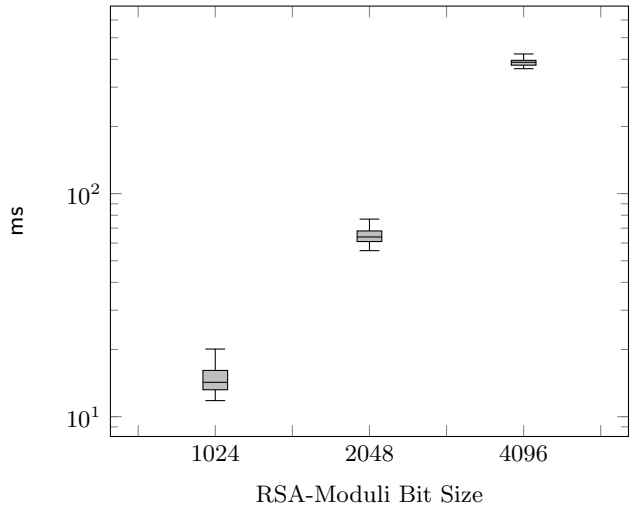


**Fig. 10.** Box-Plots of the setup protocol measurements in ms for the server.

our full setup protocol would take around four seconds on average including network delay.

*Signing Protocol.* For each signing request, we used the correct password, i.e., no "failed" signing attempts where measured. Additionally to the *sid* now also a query identifier *qid* is given as input, which we assume to be generated by an external protocol. The box-plots are in Figure 12 and Figure 13. The corresponding percentiles are depicted in Table 4 and Table 5.

The figures show that the time required for signing is — from a practical perspective — nearly constant for each security parameter. Including the communication overhead, our protocol is still faster than a real smart card, which requires at least one second for signing with a 2,048 Bit key.

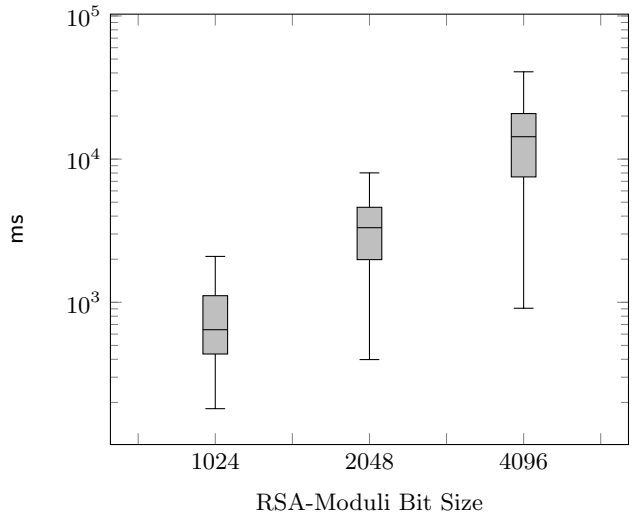


**Fig. 11.** Box-Plots of the setup protocol measurements in ms for the device.

| | 1,024Bit | 2,048Bit | 4,096Bit |
|---|---|---|---|
| Min.: | 16.49 | 73.47 | 470.30 |
| 25th Percentile: | 18.06 | 78.44 | 478.84 |
| Median: | 19.08 | 79.83 | 482.61 |
| 75th Percentile: | 20.64 | 82.38 | 487.69 |
| 90th Percentile: | 22.25 | 92.19 | 671.13 |
| 95th Percentile: | 23.91 | 113.66 | 883.83 |
| Max.: | 42.80 | 138.18 | 4′994.40 |
| Average: | 19.79 | 83.40 | 574.41 |

**Table 4.** Percentiles for signing in ms for the device.

| | 1,024Bit | 2,048Bit | 4,096Bit |
|---|---|---|---|
| Min.: | 10.90 | 62.43 | 452.01 |
| 25th Percentile: | 11.32 | 63.56 | 454.08 |
| Median: | 11.76 | 64.53 | 456.38 |
| 75th Percentile: | 12.60 | 66.58 | 470.95 |
| 90th Percentile: | 14.41 | 68.39 | 501.75 |
| 95th Percentile: | 14.96 | 72.06 | 521.64 |
| Max.: | 27.77 | 78.98 | 578.58 |
| Average: | 12.31 | 65.50 | 466.73 |

**Table 5.** Percentiles for signing in ms for the server.

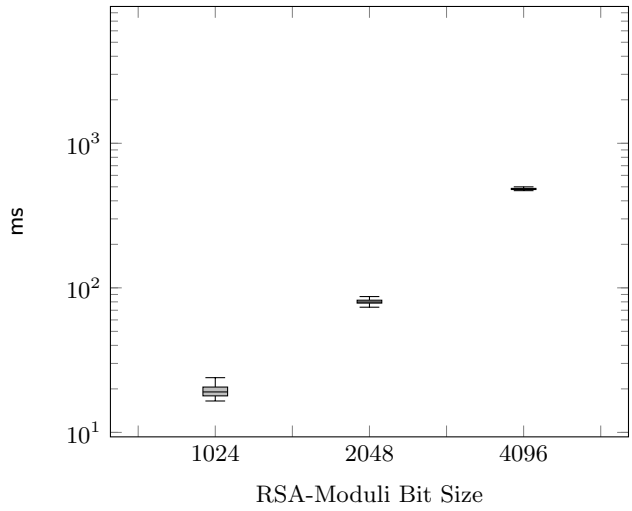


**Fig. 12.** Box-Plots of the signing protocol measurements in ms for the device.

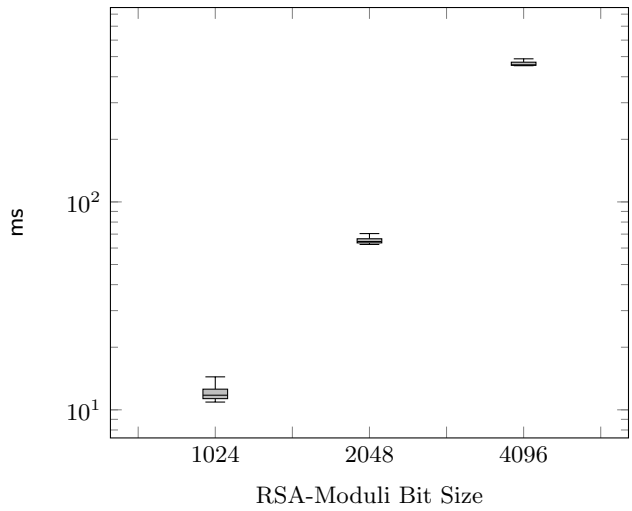


**Fig. 13.** Box-Plots of the signing protocol measurements in ms for the server.