# On the Theory and Practice of Personal Digital Signatures

Ivan Damgård and Gert Læssøe Mikkelsen*

Depertment of Computer Science, Aarhus University

**Abstract.** We take a step towards a more realistic modeling of personal digital signatures, where a human user, his mobile equipment, his PC and a server are all considered as independent players in the protocol, and where only the human user is assumed incorruptible. We then propose a protocol for issuing digital signatures on behalf of the user. This protocol is proactively UC-secure assuming at most one player is corrupted in every operational phase. In more practical terms, this means that one can securely sign using terminals (PC's) that are not necessarily trusted, as long as the mobile unit and the PC are not both corrupted at the same time. In other words, our solution cannot be broken by phising or key-logging via the PC. The protocol allows for mobile units with very small computing power by securely outsourcing computation to the PC and also allows usage of any PC that can communicate properly. Finally, we report on the results of a prototype implementation of our solution.

## 1 Introduction

When cryptographic protocols make use of digital signatures, this is usually described in the cryptographic theory literature by saying something of the following form: "Player $P_i$ signs the message $m$ using his secret key $sk_i$, and sends $m$ and the signature $\sigma_i = S_{sk_i}(m)$ to player $P_j$".

While this may be a convenient abstraction in some cases, it hides some details that are often very important in practice: in real life a "player" is usually not really a single entity but consists in fact of a human user as well as one or more computing devices he uses in order to store the key and issue the signature. Each of the devices could be corrupted by an adversary without the user being dishonest. In such a case, the theoretic model above would have to consider the entire "player" to be corrupt and would conclude that we can now no longer protect the secret key. However, in real life, it is of obvious interest to protect the user, even if some of his equipment is corrupt.

A well known example of this is the so called man-in-the-middle (or man-in-the-browser) attack which in the context of signatures takes the form of showing the user a message on screen that he would approve, while at the same time trying to have the secret key applied to a different message. If the user's secret

* Supported by the Danish Strategic research Council

key resides on his PC, protected say, by encryption under a password, a man-in-middle attack is always possible if the PC is corrupt.

In this paper, we first propose a model that we believe reflects in a more realistic way the issues arising when a private person wants to issue digital signatures using current technology. The main players are: a human user $U$, a mobile unit $M$, a terminal $T$, and a server $S$. In practice, $M$ could be a PDA, a cell-phone or special-purpose device, $T$ could be a PC (but not necessarily the user's own machine), and $S$ could be some server where the user has an account. $S$ could be the user's own machine, or it could be run by a company handling many users – the only assumption we make is that $S$ is on-line whenever a signature is to be issued. In this model, only the user is assumed incorruptible.

We assume that $T$ can interact with $U$ in a standard way via keyboard and screen. $M$ has a screen where it can show a message to be signed and may receive an OK or a reject from the user.

In practice, we would like our solutions to be mobile, i.e., any machine can in principle be used as $T$, so we therefore consider only protocols where no user specific key material is stored permanently on $T$. Another practical issue is that a mobile handheld unit can easily be lost or stolen, and it should be possible to securely replace it without having to generate keys (and issue certificates) again. This requires our solution to be secure against an adversary who first steals $M$ and later breaks into $T$ or $S$.

This issue makes it natural for us to aim for proactive security in the UC model. In proactive security, first introduced in [17], one divides time into *operational phases*, interleaved with (short) *refreshment phases*. In operational phases, the system provides normal service, while refreshment phases are typically used to update key material using fresh randomness. The adversary is assumed to only corrupt a certain number of players in every operational phase, but the set of corrupted players can be different in different phases, meaning that all players may have been corrupt at some point, and the system must still be secure. Our (adaptive) adversary may in each operational phase actively corrupt at most one of $M, T, S$. The adversary mentioned above who first steals $M$ and later breaks into $T$ or $S$ can be modeled in the proactive framework as an adversary who corrupts $M$ in one operational phase and $T$ or $S$ in the next. Note also that $T$ models any machine(s) that the user $U$ uses as terminal. So if $U$ in real life first uses an untrusted terminal in some Internet cafe and then returns to an uncorrupted PC, this means in our model that $T$ is first corrupted and then becomes honest again.

We stress that our protocols are secure, even if corruption does not take the form of loss of a device: whenever our system is operational, it is secure if the adversary has corrupted at most one of $M, T$ and $S$, even if the user is not aware of the corruption. In particular this means we are secure against phising or key-logging since this corresponds to corruption of $T$. On the the other hand if we *know* that $M$ or $S$ have been corrupted, we can make the system operational again, by replacing e.g., a lost $M$ by a new uncorrupted device and restoring the key from a backup.

We model this by introducing a new player, a database, $D$. This player is only active in the refreshment phase and is used to restore data held by $S$ that could have been erased or corrupted by an active attack. We allow the adversary to corrupt $D$ passively. In practice, we think of $D$ as run by the same party who runs $S$, and as such the introduction of $D$ models the assumption that the server's organization is able to ensure a backup that is reliable, but not necessarily secret.

The functionality we aim to implement is basically the standard UC functionality for secure signatures, except that the adversary is allowed to stop a signature from being generated [1], and a message is only signed if the user approves it. We then propose a protocol that is secure in this model. The protocol has the following properties:

- To execute the protocol, a user first needs to form the message $m$ to be signed while interacting with $T$ (one may think of using machine $T$ to buy something on the net, where $m$ is a payment order). To sign, he first authenticates himself towards $S$ (typically by entering his log-in data on $T$), and second he is shown on $M$'s screen the message $m$ and must in response tell $M$ "OK" or "reject".
- To execute the proactive refreshment phase, a user just has to update his log-in data for $S$, and if $M$ is a new mobile unit (a replacement for a stolen one), he must enter a special code on $M$ (this can be done without many key strokes, e.g., using the camera in a mobile phone to scan a 2-d bar code)
- The protocol can produce as output standard hash-and-sign RSA signatures, compatible with existing PKI's.
- The protocol allows $M$ to use only very little computing power. We can securely outsource most of the computation to $T$, so that for each signature, $M$ only has to evaluate one pseudo random function and do one addition of large numbers. This is useful if $M$ is a cheap special-purpose device, or if one wants to run a high-level language implementation on $M$ - this allows to cover many types of mobile phones using pure Java, for instance, but it will typically be much slower than device specific code.
- If desired, the protocol can keep the message to be signed secret from $S$ with no loss of efficiency. Note that this can be desirable for privacy related reasons, but could also be undesirable if one wants $S$ to keep a log of what was signed.

On the technical side, we start by borrowing a standard technique from threshold signatures where we share the secret RSA exponent additively between $M$ and $S$. We then augment this with a new technique allowing the outsourcing to $T$ mentioned above. We also propose an extension of the proactive security model by introducing two kinds of refreshment protocols: one that is done routinely with no other user intervention than a change of password, and one that is invoked in case an attack has been detected, e.g., $M$ has been lost or stolen or a virus attack

[1] It is easy to see that we cannot avoid this in a scenario where only $M$ and $S$ can store key material on-line and either of them may be corrupted.

was detected in $S$. In such cases we may have lost the information stored on $M$ or $S$, which means that the secret key is effectively lost as well. We therefore need to design a way to use back-up information stored off-line to securely reestablish the secret key. Finally, keeping the signed message secret from $S$ can be done using standard blinding techniques [8].

In the final part of the paper, we report on results from a prototype implementation of our protocol. In the prototype $M$ was a mobile phone, running a Java application while communicating with $T$ (a PC) via Bluetooth. $T$ then communicates with $S$ via the Internet and SSL. The results show that our outsourcing technique can give a significant speedup, and provide a far better experience for the user.

### 1.1 Related Work

We are not aware of any previous work that attempts to model our scenario in the UC framework. However, the idea of using a personal (mobile) device to improve security in practice has been studied in several previous papers. In [18], Parno et al. use a mobile phone to set up secure SSL/TLS connections and in [15], Mannan and Oorschot use a personal device to improve security of password authentication. Both solutions basically aim to do user authentication with improved security, in particular to protect against key-logging and phising. In [20], Weigold et al. use a trusted mobile USB device with a display and two buttons to improve security of online services such as Internet banking. All communication between the used PC and the server is routed through this trusted device, where the user has to accept sensitive transactions. [15, 20] also contains a good overview of other existing anti-phising techniques and their properties. Finally, there are many examples of using secure devices for transaction authentication, see [2, 13], for instance.

The main difference to our results is that the previous works need to assume that the personal device $M$ is uncorrupted (malware-free), while our solution is secure even if $M$ is corrupt, as long as $S, T$ are honest, due to the sharing of the key between $M$ and $S$. Also, previous works typically does not consider proactive techniques.

As for existing cryptographic techniques, previous literature considers several types of solutions that protect a secret signature key, even if one or more entities are corrupt. A first such technique is known as *Threshold Signature Schemes*, where the secret key is shared among a set of entities called signature servers. One can then have protocols that guarantee security if a majority of servers remain honest. A large body of literature exists on threshold signatures see, e.g., [1, 9, 12, 19]. In these protocols the signature servers play symmetric roles, i.e., they all execute essentially the same program and each server is assumed to be equally hard to break into (hence the honest majority assumption). Protocols for threshold signatures usually assume that the honest servers already agree on the message to be signed and take it from there without considering how such an agreement would be reached in practice.

Therefore, standard threshold signatures are not immediately applicable in our case where a private citizen wants to use digital signatures: he may store key material in different devices with completely different security properties, for instance, on a mobile unit such as a PDA or a cellphone, or on a server. Also, the idea that all players should agree on the message to be signed, does not really make sense here: we clearly want that only the user decides, and only messages approved by the user get signed. It is not even clear that we want all players to know the messages signed. If a server handling many users is involved, it may be undesirable for privacy reasons that the server knows what is signed.

A second related class of solutions is known as *Key Insulated* and *Intrusion Resilient* signatures [11, 14]. In a nutshell, this model and ours are incomparable, but we believe that our approach is the more realistic of the two.

In more detail, while we try to improve security against key exposures by requiring participation of several entities, KI/IR signatures insist that one entity that they call "the user" can sign on its own. Those schemes then instead try to limit the effect of key exposures, by making the users private key valid only for a certain period. When a period expires, the secret key must be updated using a message from a second entity, called the *key base*. Security properties generally hold assuming that user and key base are not simultaneously compromised, so the two trust models are comparable in that only one entity at a time is assumed corrupt. The properties obtained are different, however: If the assumption on our adversary holds, no signature can be generated other than those the user approves, while compromising the user in KI/IR signatures allows generating false signatures in the current period. On the other hand, KI/IR signatures may retain forward security even if both units are compromised in the same period.

The known KI/IR signatures schemes are specially engineered to get the desired security properties. Therefore, certification authorities and receivers of signatures must be aware of the scheme and its special properties to use it. Our scheme outputs completely standard hash-and-sign RSA signatures and so it can co-exist under the same PKI with any other solutions for storing the secret key. This is a very important feature for such a scheme to be useful in a real-life application. A final comment is that existing work on KI/IR signatures assumes that the "user" holds and uses a secret key, and hence ignore the problems stemming from the fact that in real life the secret key must of course be held and operated by some device that is separate from the human user.

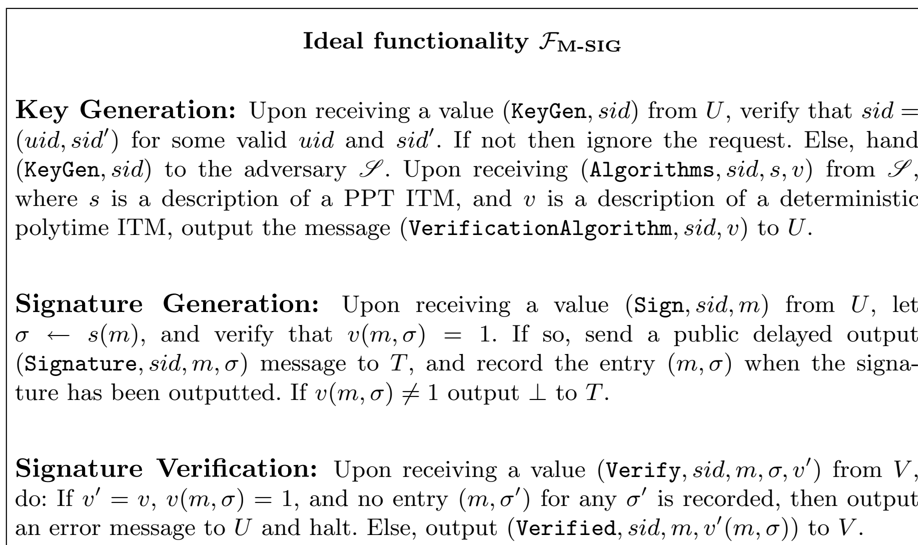
## 2 Security Model

Traditional formal models of digital signatures, e.g., the one described by Canetti in the description of the UC framework [6], are models of the computer used during signature generation and its security. Intuitively, the security we aim for is different, namely: *Only messages accepted by the human user should be signed.*

Modeling human behavior in the UC framework is not an obvious task. Modeling the human ability to decide whether a message should be accepted or rejected would result in an extremely rough approximation, and make our model

unclear. Instead we let the environment $\mathcal{Z}$ decide by sending acceptable messages to the model of the human user $U$, and a protocol is deemed secure if it only outputs signatures on messages that were given to $U$ by $\mathcal{Z}$. Since human users cannot calculate digital signatures and therefore rely on corruptable computing equipment, in our case the terminal $T$, the output of our ideal functionality is output through $T$.

The ideal functionality for our mobile signatures $\mathcal{F}_{\text{M-SIG}}$ (Fig. 1) is an extension of the functionality $\mathcal{F}_{\text{SIG}}$ by Canetti [6, Section 7.2.1]. The signer, player $S$ in $\mathcal{F}_{\text{SIG}}$ has been split into: $U$, $S$, $M$ and $T$ in our protocol, while the verifier $V$ is the same in both protocols. $\mathcal{F}_{\text{M-SIG}}$ differs from from $\mathcal{F}_{\text{SIG}}$ in that: 1) The message $m$ to be signed has to be sent to $U$ and 2) The adversary is able to stop the generation of a signature (in the model, he can do so by stopping delivery of output from $\mathcal{F}_{\text{M-SIG}}$). The main idea behind $\mathcal{F}_{\text{M-SIG}}$ is to not specify a particular signature algorithm, but to keep track of messages that have been submitted for signing and accept only these messages as signed. This is the reason for the signature verification part of $\mathcal{F}_{\text{M-SIG}}$, which at first may seem a bit counter-intuitive.


**Ideal functionality $\mathcal{F}_{\text{M-SIG}}$**

**Key Generation:** Upon receiving a value (`KeyGen`, $sid$) from $U$, verify that $sid = (uid, sid')$ for some valid $uid$ and $sid'$. If not then ignore the request. Else, hand (`KeyGen`, $sid$) to the adversary $\mathscr{S}$. Upon receiving (`Algorithms`, $sid, s, v$) from $\mathscr{S}$, where $s$ is a description of a PPT ITM, and $v$ is a description of a deterministic polytime ITM, output the message (`VerificationAlgorithm`, $sid, v$) to $U$.

**Signature Generation:** Upon receiving a value (`Sign`, $sid, m$) from $U$, let $\sigma \leftarrow s(m)$, and verify that $v(m, \sigma) = 1$. If so, send a public delayed output (`Signature`, $sid, m, \sigma$) message to $T$, and record the entry $(m, \sigma)$ when the signature has been outputted. If $v(m, \sigma) \neq 1$ output $\perp$ to $T$.

**Signature Verification:** Upon receiving a value (`Verify`, $sid, m, \sigma, v'$) from $V$, do: If $v' = v$, $v(m, \sigma) = 1$, and no entry $(m, \sigma')$ for any $\sigma'$ is recorded, then output an error message to $U$ and halt. Else, output (`Verified`, $sid, m, v'(m, \sigma)$) to $V$.

**Fig. 1.** Ideal functionality for *"secure mobile digital signatures"*, based on $\mathcal{F}_{\text{SIG}}$ in [6].

## 3 Protocol Securely Realizing $\mathcal{F}_{\text{M-SIG}}$

The main player in our protocol is the human user $U$. The main idea behind the protocol is to protect the user from a corrupt terminal or a corrupt mobile unit by letting him accept the message to be signed on both the terminal and on the

mobile device; and to ensure that the signature cannot be generated unless both units received an accept from $U$. This can be implemented by secret sharing the private exponent $d$ of the user's private key, with a simple additive secret sharing.

**Secret sharing:** The following additive sharing scheme is used to secret share the user's secret RSA key $sk = \langle d, N \rangle$: The private exponent $d$ is shared to a uniform randomly chosen $d_1$ and a value $d_2$ s.t.:

$$d \equiv d_1 + d_2 \bmod \varphi(N) \tag{1}$$

Hence, for any $m$, we have:

$$m^d \bmod N = m^{d_1} m^{d_2} \bmod N \tag{2}$$

Note that (2) still holds if the addition in (1) is done over the integers.

By giving $d_1$ to the mobile device $M$ and $d_2$ to the server $S$, a simple protocol realizing $\mathcal{F}_{\text{M-SIG}}$ can be implemented. In this protocol the message, if accepted by the user on the terminal $T$ is sent to $M$ and to $S$. Then $M$ shows the message on its screen and will sign the message with its exponent share if the user accepts the message. $S$ will sign the message with the other exponent share if the correct password is typed into $T$ and sent to $S$. Finally, $T$ can assemble the complete signature by multiplying the two "half signatures".

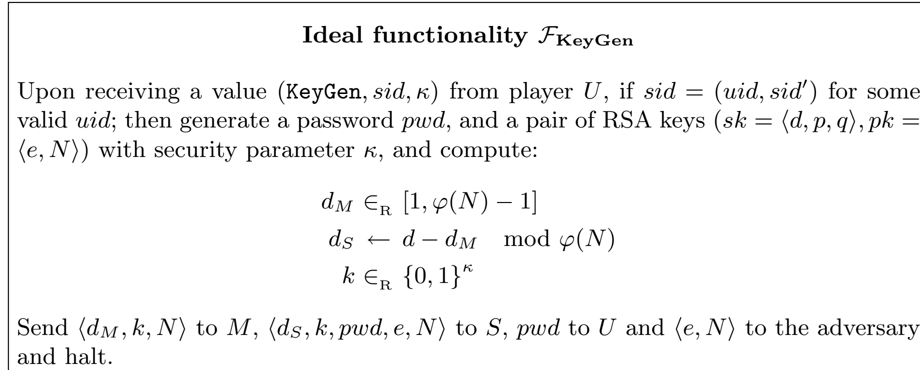
This protocol, however, requires $M$ to do a full scale exponentiation. This is problematic because we want to include cases where $M$ is a special purpose, cheap and small device, or where the software running on $M$ is high-level code only, for portability (such as pure Java on a mobile phone). The Chinese Remainder Theorem (CRT) method[2] often used to speed up RSA exponentiation cannot be used here, since this would reveal the factorization of $N$ to the mobile device. Alternatively, we could make $M$'s exponent share be a number (much) smaller than $d$. This would speed up $M$'s computation, but would reveal significant information to $S$ about $d$. We do not know if this is secure, but we strongly suspect it is not. We therefore propose to exploit the fact that $T$ is likely to have much more computing power than $M$. Doing this requires some changes in the protocol, as explained in the following section.

### 3.1 Protocol $\pi_{\text{M-SIG}}$, for Computationally Limited $M$

$\pi_{\text{M-SIG}}$ assumes keys are set up beforehand, this is done by using an ideal functionality $\mathcal{F}_{\text{KeyGen}}$ (Fig. 2) for generation and distribution of keys and password. Because the signatures generated follow existing standards, verification is normal RSA signature verification without any communication involved. This means that the main part of $\pi_{\text{M-SIG}}$ is generation of signatures.

[2] For a description of the CRT speedup method see [3, 5.2]

**Key Generation** $\mathcal{F}_{\text{KeyGen}}$ Will generate a password *pwd* for the user, generate keys, and share the user's private key. After *pwd* and keys have been generated they are distribute to the respective players.



**Ideal functionality $\mathcal{F}_{\textbf{KeyGen}}$**

Upon receiving a value $(\texttt{KeyGen}, sid, \kappa)$ from player $U$, if $sid = (uid, sid')$ for some valid *uid*; then generate a password *pwd*, and a pair of RSA keys $(sk = \langle d, p, q \rangle, pk = \langle e, N \rangle)$ with security parameter $\kappa$, and compute:

$$\begin{aligned} d_M &\in_{\text{R}} [1, \varphi(N) - 1] \\ d_S &\leftarrow d - d_M \mod \varphi(N) \\ k &\in_{\text{R}} \{0,1\}^{\kappa} \end{aligned}$$

Send $\langle d_M, k, N \rangle$ to $M$, $\langle d_S, k, pwd, e, N \rangle$ to $S$, *pwd* to $U$ and $\langle e, N \rangle$ to the adversary and halt.

**Fig. 2.** Functionality $\mathcal{F}_{\text{KeyGen}}$. Generating keys and a password for the user.

**Signature Generation** When $U$ receives a value $(\texttt{Sign}, sid, m)$, $m$ is forwarded to $T$ together with the password *pwd*, then $T$ starts a signing protocol where $U$ is asked from $M$ if $m_M$ should be signed. $U$ will accept message $m_M$ from $M$ iff $m_M = m$. Before the formal definition of $\pi_{\text{M-SIG}}$ we need to define some components of the protocol.

**Definition 1.** $\mathcal{H}(m)$ *denotes the hashing and padding applied to the message* $m$ *before the exponentiation is done in the used RSA signature scheme.*

Hashing and padding is needed to make RSA signature schemes secure against *chosen plaintext attacks*, and is therefore already used in most standards. Our protocol is secure no matter how $\mathcal{H}$ works, as long as combining $\mathcal{H}$ and RSA results in a secure signature scheme. Hashing can also give some amount of privacy because the server only needs to see $\mathcal{H}(m)$ and not $m$ itself. If desired, blinding can be applied to ensure the users privacy unconditionally, see section 6. If logging is desired on the server, the server can do the hashing and padding and $m$ itself can be sent around in the protocol.

**Definition 2.** $F_k(\cdot)$ *denotes a* secure pseudo-random function *with* $\widetilde{\kappa} + \kappa$*-bit output and key* $k$*, with* $\tilde{\kappa}$ *being the length of the RSA keys. More precisely, a polynomial time bounded adversary who gets oracle access to either* $F_k()$ *or a random function cannot distinguish the two alternatives with an advantage that is non-negligible (in the length of* $k$*).*

An overview of the protocol can be found in Fig. 3. In case of errors during execution of the signing protocol, players communicating with the terminal $T$ will

send $\perp$ to $T$ and $T$ will then stop the protocol and return $\perp$ to the environment $\mathcal{Z}$. The protocol is executed the following way: First the message $m$ is sent to the user $U$ from $\mathcal{Z}$, and $U$ sends $m$ to $T$ together with *pwd*. $T$ will now send $m$ to the mobile device $M$. $M$ sends $m_M$ ($m_M \leftarrow m$) to $U$ and $U$ returns (`accept`) if $m_M = m$, else $U$ rejects and $\perp$ is returned to $M$ and forwarded to $T$. If $U$ accepts, $M$ calculates $\delta_M$ and sends $\delta_M$ to $T$.

$$\delta_M \leftarrow F_k(\mathcal{H}(m)) + d_M \tag{3}$$

The value $\delta_M$ is a blinding of the key share $d_M$ known to $M$. Because $k$ is unknown to $T$, $T$ can do the exponentiation without gaining knowledge of $d_M$ (the blinding is later removed by $S$).

When $T$ receives $\delta_M$, $T$ calculates $\sigma_M$ and $\mathcal{H}(m)$ and send these values and *pwd* to $S$.

$$\sigma_M \leftarrow \mathcal{H}(m)^{\delta_m} \mod N \tag{4}$$

When $S$ receives $\sigma_M$, $\mathcal{H}(m)$ and *pwd* it checks if *pwd* is correct, if not $\perp$ is sent back to $T$. Else $\sigma_S$ (6) and $\sigma$ (7) are calculated and $S$ checks if $\sigma$ is a valid signature of $\mathcal{H}(m)$, if this is the case $\sigma$ is sent back to $T$, if not $\perp$ is sent back. Note that sending $\sigma_M$ to $S$ lets $S$ calculate and verify $\sigma$, and thus indirectly check that $m$ has been accepted by $U$ through both $T$ and $M$. The protocol *might* be secure without this check, however, our proof requires it. Furthermore when we later extend the protocol to be proactive, this check make some recoveries simpler and thereby more user friendly.

$$\delta_S \leftarrow d_S - F_k(\mathcal{H}(m)) \tag{5}$$

$$\sigma_S \leftarrow \mathcal{H}(m)^{\delta_S} \mod N \tag{6}$$

$$\sigma \leftarrow \sigma_M \times \sigma_S \mod N \tag{7}$$

$$= \mathcal{H}(m)^{d_M + F_k(\mathcal{H}(m)) + d_S - F_k(\mathcal{H}(m))} \mod N \tag{8}$$

$$= \mathcal{H}(m)^d \mod N \tag{9}$$

**Communication:** The model we use assumes communication to be secret from the adversary unless he has corrupted one of the communicating parties. The real-life justification for this differs between the different communication channels used. For key generation the ideal functionality $\mathcal{F}_{\text{KeyGen}}$ is used. Formally, communication with ideal functionalities is done over perfect secure channels and the functionality specifies what to leak to the adversary. $\mathcal{F}_{\text{KeyGen}}$ leaks the public key and the fact that keys have been generated. $\mathcal{F}_{\text{KeyGen}}$ is thought of either: as a trusted third player, in which case secure encrypted communication is a reasonable assumption; or alternatively as a protocol doing secure shared key generation e.g., [4], in which case communication to the protocol is done locally.

During signature generation different communication channels are used, these channels are modeled by an ideal functionality "Secure message transmission" $\mathcal{F}_{\text{SMT}}$ delivering the message $n$ and leaking the length of $n$ to the adversary. For

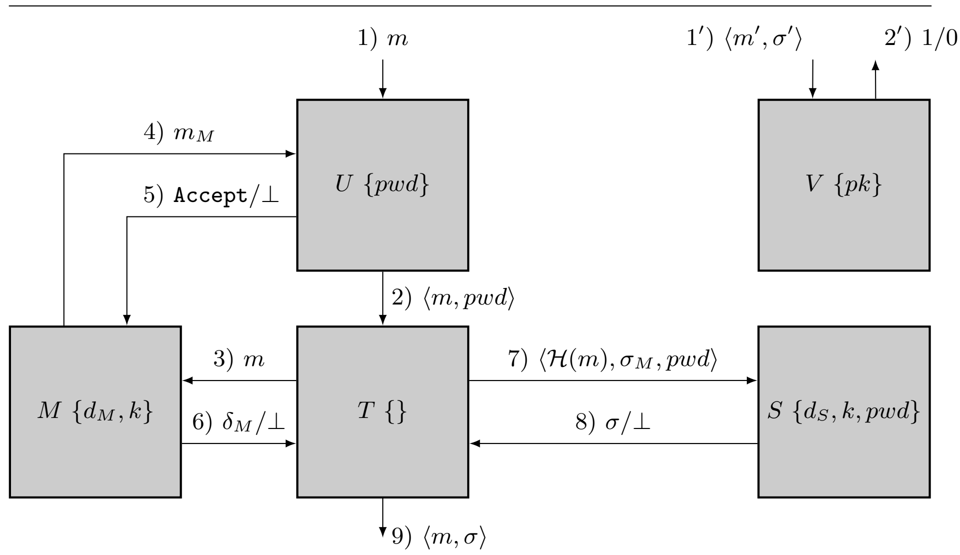



**Fig. 3.** Overview of the signing phase in $\pi_{\text{M-SIG}}$. For simplicity *sid* is left out of all messages in this overview. Values sent to the players during key generation are presented after the name of the player.

a concrete formal definition of $\mathcal{F}_{\text{SMT}}$ see [6, section 6.3]. Communication that involves $U$ models what the user sees and types on the terminal or mobile device and is therefore assumed secure against adversaries located physically away.

Communication between $T$ and $S$ is done over a cryptographically secured channel (but it only has to be secure if $T$ and $S$ are honest). This can be done using SSL/TLS if an appropriate public-key infrastructure is in place, but a password-based key exchange such as SRP or a password-based cipher suite for TLS [5] is a more natural and secure solution (as this avoids problems like selection of the right certificate to use for obtaining $S$'s public key).

For communication between $M$ and $T$, there are two possible justifications for assuming it to be secure: 1) Communication happens over a secure connection, this could be via USB cable or a connection where security is based on cryptography; the latter case can be feasible even if $M$ is computationally weak, namely if RSA with a small public exponent is used, or in case we use a secure Bluetooth protocol with pairing. 2) We could also base ourselves on the fact that the communication only has to be secure if $M, T$ are honest and $S$ is corrupt. Since $S$ is typically located physically away from $T$ and $M$, one may decide that unencrypted communication is good enough if done such that it can only be picked up in physical proximity.

## 4 Protocol $\pi_{\text{M-SIG}}$ UC-realizes $\mathcal{F}_{\text{M-SIG}}$

In this section we will prove that under appropriate assumptions $\pi_{\text{M-SIG}}$ UC-realizes $\mathcal{F}_{\text{M-SIG}}$. The security of $\pi_{\text{M-SIG}}$ is obviously based on the security of the underlying RSA signature scheme, hence we need:

**Assumption 1.** *With proper choice of hashing and padding function $\mathcal{H}(\cdot)$, the underlying RSA signature scheme: $Sig(m) = \mathcal{H}(\cdot)^d \mod N$ is secure against* adaptive chosen plaintext attacks.

Our protocol clearly needs that $U$ can securely authenticate himself towards $S$. For concreteness, we have specified that this happens using a password *pwd*, but in fact any authentication method could be used, as long as it is secure against an adversary that does not corrupt $S$ or $T$. We have chosen to leave out the details of the authentication and its security by simply assuming that the adversary cannot with significant probability get the password except by corrupting a player who has seen it. Actually, since the adversary has to do online attacks if trying to guess *pwd*, and the server implementation can take this into account, this assumption may be justifiable. Other ways and discussions about modeling password security in the UC framework can be found in [7].

**Assumption 2.** *If the adversary does not corrupt $S$ or $T$, he can produce the correct password with only negligible probability.*

**Theorem 1 ($\pi_{\text{M-SIG}}$ UC-realizes $\mathcal{F}_{\text{M-SIG}}$).** *Under assumptions 1 and 2 and if $F_k$ is secure, $\pi_{M\text{-}SIG}$ UC-realizes $\mathcal{F}_{M\text{-}SIG}$ with respect to adaptive and active adversaries, corrupting at most one of the players: $T$, $M$ or $S$.*

*Proof.* For any real world adversary $\mathcal{A}$ interacting with $\pi_{\text{M-SIG}}$, and corrupting at most one of the players: $T$, $M$ or $S$, we need to show that there exists a simulator $\mathscr{S}$ interacting with $\mathcal{F}_{\text{M-SIG}}$, such that no PPT environment $\mathcal{Z}$ can distinguish $\mathcal{A}$ interacting with $\pi_{\text{M-SIG}}$ from $\mathscr{S}$ interacting with $\mathcal{F}_{\text{M-SIG}}$. The proof of this is done in two steps: First we present an $\mathscr{S}$ capable of simulating $\pi_{\text{M-SIG}}$ for all $\mathcal{A}$, except if $\mathcal{Z}$ is able to produce a forged signature (i.e., a signed message not accepted by the user). Next we present a reduction, which, given a $\mathcal{Z}$ capable of forging signatures, can use $\mathcal{Z}$ to forge "normal" RSA signatures, and thereby breaking assumption 1. Consequently, simulation only fails with negligible probability.

**Simulating $\pi_{\text{M-SIG}}$** The simulator $\mathscr{S}$ needs to simulate the adversary's view of $\pi_{\text{M-SIG}}$, when the players forwards all input to $\mathcal{F}_{\text{M-SIG}}$ instead of running $\pi_{\text{M-SIG}}$. $\mathscr{S}$ has to simulate the leakage (i.e., the length of the sent data) of communication. If a player is corrupted, $\mathscr{S}$ in addition has to simulate the view of this player. $\mathscr{S}$ will do this by generating keys following the algorithm of $\mathcal{F}_{\text{KeyGen}}$ and sending the expected keyshares to corrupt players. Since it knows all secret keys, it can now simulate $\pi_{\text{M-SIG}}$ by simply running the protocol. It is evident that verification of signatures is the only way for $\mathcal{Z}$ to distinguish

simulation from the real protocol. Invalid signatures will be rejected both in the real and the ideal world, genuine valid signatures will be accepted in both worlds. However, forged signatures will only be accepted in the real world since $\mathcal{F}_{\text{M-SIG}}$ enforces unforgeability. Thus $\mathscr{S}$ simulates $\pi_{\text{M-SIG}}$ perfectly except if the environment $\mathcal{Z}$ is able to produce a forged signature.

**Reduction** To prove that $\pi_{\text{M-SIG}}$ provides unforgeability, we construct a reduction that - if $\pi_{\text{M-SIG}}$ is insecure - can beak the underlying RSA signatures scheme, and thereby violate assumption 1. The idea is that if there exist a PPT environment $\mathcal{Z}$ that with nonnegligble probability can forge signatures, based on information gained by controlling a corrupted player; we would be able to use $\mathcal{Z}$ to forge an RSA signature in polynomial time. The reduction $Red_{\text{RSA}}$ is formally described in Fig. 4. A forge of an RSA signature is modeled by giving $Red_{\text{RSA}}$ access to an RSA oracle $\mathcal{O}_{\text{RSA}}$. $\mathcal{O}_{\text{RSA}}$ will return a public RSA key to $Red_{\text{RSA}}$ when prompted, and sign messages when $Red_{\text{RSA}}$ sends them. We say that $Red_{\text{RSA}}$ has forged an RSA signature successfully, if $Red_{\text{RSA}}$ can output a signature on a message that has not been signed by $\mathcal{O}_{\text{RSA}}$.

We need to prove that communicating with $Red_{\text{RSA}}$ is indistinguishable from $\pi_{\text{M-SIG}}$, so $\mathcal{Z}$ will behave the same way in both cases. It is evident that simulating communication (i.e., leak the length of data sent) can be done.

If $\mathcal{A}$ corrupts $M$, $\mathcal{Z}$ learns the random variables $k$ and $d_M$. $k$ has the same distribution in both cases, while $d_M$ is uniform random in $[1, \varphi(N)-1]$ in $\pi_{\text{M-SIG}}$ and $d_M$ in $Red_{\text{RSA}}$ is uniform random in $[1, N]$. The two distributions are, however, statistically close. In both cases $\mathcal{Z}$ also learns all messages $m$ signed so far, and since neither $M$ or $T$ has been corrupted, $U$ has accepted them all. So all input is indistinguishable when corrupting $M$.

Both $U$ in $\pi_{\text{M-SIG}}$ and $Red_{\text{RSA}}$ acting as $U$ will accept only a genuine message $m$; furthermore both in $Red_{\text{RSA}}$ and in $\pi_{\text{M-SIG}}$ sending $\delta_M \equiv d_M + F_k(\mathcal{H}(m)) \mod \varphi(N)$, but nothing else, to $T$ will result in a signature. This proves that controlling the output of $M$ does not give $\mathcal{Z}$ the ability to distinguish between $\pi_{\text{M-SIG}}$ and $Red_{\text{RSA}}$.

If $\mathcal{A}$ corrupts $T$, $\mathcal{Z}$ learns $pwd$, $k$, all signed messages $m$ and there signatures $\sigma$, the distribution of these are equal in the two cases. In addition $\mathcal{Z}$ learns $\delta_M$. In $Red_{\text{RSA}}$ $\delta_M$ is uniformly chosen in $\{0,1\}^{\widetilde{\kappa}+\kappa}$, whereas in $\pi_{\text{M-SIG}}$, $\delta_M = d_M + F_k(\mathcal{H}(m))$. By definition (2) $F_k$ is indistinguishable from a uniform chosen $\widetilde{\kappa}+\kappa$ bit value, and since $d_M$ is $\widetilde{\kappa}$ bits long, the two distributions are statistically close.

When $T$ sends a message $m$ to $M$, $M$ will in both cases return a indistinguishable $\delta_M$ if $m$ did originate from $U$, while $\perp$ is returned to everything else. $T$ sending $S$ a correct triple $\langle \mathcal{H}(m), \sigma_M, pwd \rangle$ will in both cases result in $S$ returning a signature $\sigma$ on $m$, the same is the case with a correct triple $\langle \mathcal{H}(m'), \sigma'_M, pwd \rangle$ from an earlier signed message $m'$. On the contrary in $Red_{\text{RSA}}$ $T$ would get $\perp$ back, if a correct triple $\langle \mathcal{H}(\bar{m}), \bar{\sigma_M}, \bar{pwd} \rangle$ of a not yet signed message $\bar{m}$ is send to $S$, whereas $S$ in $\pi_{\text{M-SIG}}$ would produce a signature $\bar{\sigma}$ of $\bar{m}$. Since we assume that $F_k$ is secure (definition 2), the probability of $\mathcal{A}$ producing a correct $\delta_M$ and

**The reduction $Red_{\mathbf{RSA}}$**

$Red_{\text{RSA}}$ takes the following inputs: an environment $\mathcal{Z}$; the security parameter $\kappa$, the length of RSA keys $\widetilde{\kappa}$ and an RSA oracle $\mathcal{O}_{\text{RSA}}$.

**Key Generation:** Upon receiving $(\texttt{KeyGen}, sid)$ from $U$, verify that $sid = (uid, sid')$ for some valid $uid$ and $sid'$. If not then ignore the request. Else, ask $\mathcal{O}_{\text{RSA}}$ for the public RSA key $pk = \langle e, N\rangle$ and output $pk$ as $(\texttt{VerificationAlgorithm}, sid, v(pk))$ public delayed to $U$, $v(pk)$ being the verification algorithm, with public key $pk$.

**Signature Generation (all honest):** Upon receiving $(\texttt{Sign}, sid, m)$ from $U$. Send $(\texttt{Sign}, sid, m)$ to $\mathcal{O}_{\text{RSA}}$, wait for a signature $\sigma$ of $m$ from $\mathcal{O}_{\text{RSA}}$, store $\langle m, \sigma\rangle$ and output $(\texttt{Signature}, sid, m, \sigma)$ to $T$.

**Signature Verification:** Upon receiving $(\texttt{Verify}, sid, m, \sigma, v')$ from $V$, do: If $v' = v$, $v(m, \sigma) = 1$, and no entry $m$ is recorded, then output $(\texttt{RSA-Broken}, m, \sigma)$ and halt. Else, output $(\texttt{Verified}, sid, m, v'(m, \sigma))$ to $V$.

**Corruption of $M$:** Pick $d_M \in_{\text{R}} [1, N]$, $k \in_{\text{R}} \{0,1\}^{\widetilde{\kappa}+\kappa}$ and send $d_M$, $k$ and all stored messages $m$ as simulated input to $M$. From now on when $\mathcal{Z}$ sends $m$ to $U$, send $m$ to $M$. If $M$ thereafter sends $m' \neq m$ to $U$, return $\perp$ to $M$, if $M$ on the other hand sends $m$ to $U$, return $(\texttt{Accept})$ to $M$. If $M$ sends $\delta_M \equiv d_M + F_k(\mathcal{H}(m)) \mod \varphi(N)$ to $T$ after having received $m$, output $(\texttt{Signature}, sid, m, \sigma)$ to $T$, else output $\perp$ to $T$.

**Corruption of $T$:** Pick $pwd$ as $\mathcal{F}_{\text{KeyGen}}$ would, and from all stored pairs $\langle m, \sigma\rangle$ calculate the simulated input ($m$, $pwd$, $\delta_M$, $\sigma$) of $T$:

$$\delta_M \in_{\text{R}} \{0,1\}^{\widetilde{\kappa}+\kappa} \tag{10}$$

When $\mathcal{Z}$ sends $m$ to $U$, send $\langle m, pwd\rangle$ to $T$. If $T$ now sends $m' \neq m$ to $M$, return $\perp$. If $m' = m$ return a new random $\delta_M$ (10) to $T$. If $T$ now sends $\langle \mathcal{H}(m), \sigma_M, pwd\rangle$, with $\sigma_M = m^{\delta_M} \mod N$ to $S$ return the signature $\sigma$ of $m$. If $T$ at any point sends a correct triple $\langle \mathcal{H}(m'), \sigma'_M, pwd\rangle$ of a previous signed message $m'$, return the signature $\sigma'$ for $m'$. If $T$ sends anything else to $S$ return $\perp$ to $T$.

**Corruption of $S$:** Pick $d_S \in_{\text{R}} [1, N]$, $k \in_{\text{R}} \{0,1\}^{\widetilde{\kappa}+\kappa}$ and $pwd$ as $\mathcal{F}_{\text{KeyGen}}$ would. From all stored pairs $\langle m, \sigma\rangle$ calculate the simulated input ($d_S$, $k$, $\mathcal{H}(m)$, $pwd$, $\sigma_M$) of $S$:

$$\sigma_M \leftarrow \sigma \times \left(\mathcal{H}(m)^{(d_S - F_k(\mathcal{H}(m)))}\right)^{-1} \mod N \tag{11}$$

When $\mathcal{Z}$ sends $m$ to $U$, send $\langle \mathcal{H}(m), \sigma_M, pwd\rangle$ to $S$, if $\sigma$ is returned output $(\texttt{Signature}, sid, m, \sigma)$ to $T$, else output $\perp$ to $T$.

**Fig. 4.** Reduction from forgery by interacting with $\pi_{\text{M-SIG}}$ to forgery of normal RSA signatures.

thereby a correct $\sigma_M$ is, however, negligible. All other data send from $T$ to $S$ will in both cases result in $\perp$ in return. So corrupting $T$ will not let $\mathcal{Z}$ distinguish.

If $\mathcal{A}$ corrupts $S$, $\mathcal{Z}$ learns $pwd$, the hash values $\mathcal{H}(m)$ and the signatures $\sigma$ of all signed messages, the distributions of these are equal in both cases. $\mathcal{Z}$ also learns $\sigma_M$ of the signed messages which has been constructed different in the

two cases; however, if $\mathcal{H}(m)^{(d_S - F_k(\mathcal{H}(m)))}$ has an inverse[3] $\mod N$ then:

$$\sigma \equiv \sigma_M \times \sigma_S \mod N \tag{12}$$

$$\Rightarrow \sigma_M \equiv \sigma \times \sigma_S^{-1} \equiv \sigma \times \left(\mathcal{H}(m)^{(d_S - F_k(\mathcal{H}(m)))}\right)^{-1} \mod N \tag{13}$$

So $\mathcal{Z}$ cannot distinguish $Red_{\text{RSA}}$ from $\pi_{\text{M-SIG}}$ by the input to $S$. If $S$ sends a correct signature $\sigma$ this signature is the output of both $Red_{\text{RSA}}$ and $\pi_{\text{M-SIG}}$ and anything else results in $\perp$ in both cases.

This proves that under the assumptions 1, 2 and that $F_k$ is secure no PPT environment can forge signatures, and thereby we have proven Theorem 1.

## 5 Proactive Security

As pointed out in the introduction, proactive secure protocols contain alternating operational and refreshment phases, where the latter are used to refresh the stored key material.

One way to do refreshment is to update the user password and then reshare $d$ by adding a random value to one share and subtract the same value from the other share. This solution is the *Refreshment* protocol described below, and this does in fact give us proactive security. This may seem strange since the adversary can do an active attack on $M$, for instance, and delete $M$'s share of the key. Then we can never issue signatures again, but formally speaking, this is not a problem because our ideal functionality allows the adversary to stop signatures from being generated. However, in the real world, we would want to be able to get the system operational again, particularly in case $M$ (e.g., the users mobile phone) is lost or stolen. This can be thought of as a corruption of $M$, where in addition it becomes known to the honest players that $M$ has been corrupted. If such an event happens, we can exploit the knowledge that $M$ was corrupted, to replace it by a new uncorrupted device and reestablish the key sharing from a back-up. This is done in the alternative *Refreshment** protocol below.

### 5.1 Proactive Definition of Security

Technical details on how to model proactive security in the UC framework can be found in [1] and [6]. We give a short summary here: As usual the adversary may corrupt players (adaptively), but when a player is corrupted, the adversary is not given the complete history of that player (contrary to the standard case), only the history dating back to the start of the current operational phase. The adversary may decide to leave a corrupted player when a refreshment begins, and this player now again follows the protocol, starting from some default state. The adversary may then corrupt a new player in the next operational phase, as long as

[3] If not, we can construct a nontrivial factor of N, and thereby forge RSA signatures of arbitrary messages.

the number of corrupted players stays below the specified threshold. Corruption during a refreshment phase is not allowed or, better said, it counts as if the involved player is also corrupt in both the previous and following operational phases.

It is standard to let the environment decide when a refreshment phase begins by sending *refreshment* as input to all honest players. Motivated by the above discussion, we extend the model by allowing the environment to send either *refreshment*, signaling the start of a routine refreshment, or *refreshment**, signaling the start of a refreshment where a device has to be set up from scratch, but is assumed honest. We assume that the environment only sends *refreshment** if this makes sense, that is when the adversary has left a player, and we therefore can assume the players to be honest during the *refreshment** phase. This models the case where the mobile device was lost and a new, not yet corrupted one is to be set up, or when a virus attack on the server has been detected, but after clean-up and reboot, we believe the server is honest again.

### 5.2 Protocol $\pi_{\mathcal{P}\text{-M-SIG}}$, a Proactive version of $\pi_{\text{M-SIG}}$

The ideal functionality we want to implement is $\mathcal{F}_{\text{M-SIG}}$, as in previous sections. The protocol $\pi_{\mathcal{P}\text{-M-SIG}}$ is a proactive version of $\pi_{\text{M-SIG}}$. To be able to do *refreshment** we introduce a new player $D$ in the protocol. $D$ is a database with the property that the entry for user $U$ only is writable during key generation, therefore we only allow passive corruption of $D$. During key generation a backup share of $d$ is given to the human user, and during the *refreshment** phase the user has to send this share to the mobile device. The length of this share is evidently beyond the capacity of information users can enter on a keyboard. Nevertheless this can be solved by e.g., storing the share as a 2D barcode on paper, and send it to $M$ via a camera, if $M$ is a camera equipped cellphone. Another solution is to store the share on a USB-pen or similar. A third solution is to give a short key to the user and generate the share pseudo-randomly from the key. It is important to understand that the back-up share is erased from $M$ as soon as the refreshment is over, and the method is therefore secure under the assumption that a fresh mobile device will stay honest in the short time it takes the refreshment to complete.

**Formal Description of $\pi_{\mathcal{P}\text{-M-SIG}}$** Signature generation and verification is handled in the same way as in $\pi_{\text{M-SIG}}$.

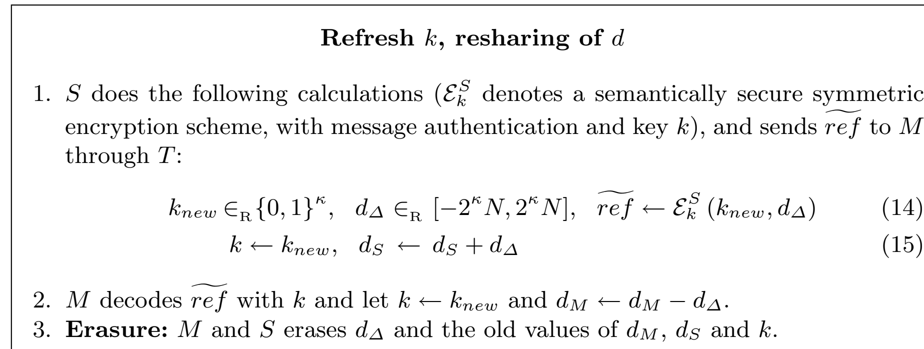
**Key Generation** Like in $\pi_{\text{M-SIG}}$ key generation is handled by an ideal functionality, $\mathcal{F}_{\mathcal{P}\text{-KeyGen}}$ is defined as $\mathcal{F}_{\text{KeyGen}}$ (Fig. 2) with the following extensions: First to simplify[4] our security proof we blind the sharing of $d$ with $d_\Delta \in_{\text{R}} [-2^\kappa N, 2^\kappa N]$ s.t. $d_S \leftarrow d_S + d_\Delta$ and $d_M \leftarrow d_M - d_\Delta$, blinding happens during the *refreshment* phase, so the small expansion of the shares is not an issue. Second a backup

[4] Blinding from start avoids treating the first operational phase as a special case.

sharing $\langle \widehat{d_M}, \widehat{d_S} \rangle$ of $d$ is computed. $\widehat{d_M}$ is sent to $U$ in addition to the values sent by $\mathcal{F}_{\text{KeyGen}}$, and $\widehat{d_S}$ is sent to, and stored by $D$. This backup sharing is required for *refreshment**.

**Communication** Communication is done with the same assumptions as in $\pi_{\text{M-SIG}}$, with an extra secure line from $S$ to $M$, which is used once during each *refreshment** phase. This line should either be thought of as provided by the mobile phone network, if $M$ is a cellphone, or a physical link between $S$ and $M$, if $M$ is some special purpose device.

**Refreshment and Refreshment*** The *Refreshment* protocol first does refresh of $k$ and resharing of $d$, see Fig. 5, and then *pwd* is updated using the protocol in in Fig. 7. To check if we are indeed in a valid state, a “test-signature” can be issued afterwards.



**Refresh $k$, resharing of $d$**

1. $S$ does the following calculations ($\mathcal{E}^S_k$ denotes a semantically secure symmetric encryption scheme, with message authentication and key $k$), and sends $\widetilde{ref}$ to $M$ through $T$:

$$k_{new} \in_{\text{R}} \{0,1\}^{\kappa}, \quad d_\Delta \in_{\text{R}} [-2^\kappa N, 2^\kappa N], \quad \widetilde{ref} \leftarrow \mathcal{E}^S_k(k_{new}, d_\Delta) \tag{14}$$

$$k \leftarrow k_{new}, \quad d_S \leftarrow d_S + d_\Delta \tag{15}$$

2. $M$ decodes $\widetilde{ref}$ with $k$ and let $k \leftarrow k_{new}$ and $d_M \leftarrow d_M - d_\Delta$.
3. **Erasure:** $M$ and $S$ erases $d_\Delta$ and the old values of $d_M$, $d_S$ and $k$.

**Fig. 5.** Refreshing $d_M$, $d_S$ and $k$ in the *refreshment* phase of $\pi_{\mathcal{P}\text{-M-SIG}}$.

The *Refreshment** protocol first creates a new value of $k$ and shares of $d$ from the backup, see Fig. 6. It then does the refreshment of *pwd* as in Fig. 7 using a blank password as the old value of *pwd* because the adversary might have changed the password if he had control of $S$ earlier.

A couple of remarks on the protocol for updating passwords: we append a 1-bit to the input of $F_k$ since then no input used in refreshment can be equal to inputs used in signature generation, so a corrupt $T$ does not get $F_k$-values that can be misused later. Using the string $s$ to hide $pwd_{new}$ is not strictly necessary in our model - since we assume a secure channel from $T$ to $S$ when both are honest, $T$ could just send $pwd_{new}$. In practice, however, if the channel is set up using password-based key exchange as discussed earlier, it is not secure if the old password has been compromised, say, by an earlier attack on $T$. Using the extra hiding under $s$ solves this problem.

**Restore $k$ and shares of $d$ from backup**

1. $D$ sends $\widehat{d_S}$ to $S$. $S$ chooses $k \in_R \{0,1\}^\kappa$ and $d_\Delta \in_R [-2^\kappa N, 2^\kappa N]$, sets $d_S \leftarrow \widehat{d_S} + d_\Delta$ and sends $\langle k, d_\Delta \rangle$ to $M$.
2. $M$ sets $k$ to the received value and sends (`send-backup`) to $U$.
3. $U$ returns $\widehat{d_M}$ to $M$ and $M$ sets $d_M \leftarrow \widehat{d_M} - d_\Delta$.
4. The protocol for refreshment of $pwd$ (Fig. 7) is run, with old password $pwd$ being blank.
5. **Erasure:** $M$ and $S$ erases $\widehat{d_M}$, $\widehat{d_S}$, $d_\Delta$ and the old values of $d_M$, $d_S$ and $k$.

**Fig. 6.** *Refreshment** phase of $\pi_{\mathcal{P}\text{-M-SIG}}$.

**Refreshment of** $pwd$

1. $U$ starts by sending a request $(\texttt{refresh}, pwd, pwd_{new})$ to $T$.
2. $T$ computes a challenge $c \leftarrow \mathcal{H}(pwd, pwd_{new}, r)$, and $r \in_R \{0,1\}^\kappa$, $s \in_R \{0,1\}^\ell$ ($\ell$ is the length of $pwd_{new}$) and sends $(\texttt{ch-pwd}, c), s$ to $M$.
3. $M$ sends (`ch-pwd?`) to $U$, and $U$ returns either $\perp$ or (`OK`) to $M$.
4. If $\perp$ was returned to $M$, $\perp$ is forwarded to $T$, while in case of (`OK`), $M$ calculates a response $\rho \leftarrow F_k(c|1)$, where $(c|1)$ means $c$ concatenated with a 1-bit, and $\alpha = \mathcal{E}_k^S(s)$. It sends $\rho, \alpha$ to $T$.
5. $T$ sends $(\texttt{ch-pwd}, pwd, pwd_{new} \oplus s, \alpha, r, \rho)$ to $S$.
6. $S$ decrypts *alpha* and calculates $pwd_{new} = (pwd_{new} \oplus s) \oplus s$. If $S$ accepts $pwd$, and if $\rho = F_k(\mathcal{H}(pwd, pwd_{new}, r))$, then $S$ will set $pwd \leftarrow pwd_{new}$.

**Fig. 7.** Refreshment of the password $pwd$ in $\pi_{\mathcal{P}\text{-M-SIG}}$.

### 5.3 Security of $\pi_{\mathcal{P}\text{-M-SIG}}$

In this section we prove in theorem 2, the security of $\pi_{\mathcal{P}\text{-M-SIG}}$, however, we also comment on, and emphasize some of the security properties of $\pi_{\mathcal{P}\text{-M-SIG}}$, since not all are covered directly by the UC framework and theorem 2. The proof of Theorem 2 is analogous to the one for theorem 1 and can be found in the full version of this paper [10]. The only substantial change is that the reduction showing that the protocol does not allow forgery of signatures, now needs to simulate refreshment phases without knowing the secret key, which turns out to be straightforward. Theorem 2 relies on our earlier assumptions, but also requires a standard assumption of the security of cryptographic hash functions:

**Assumption 3.** *For any PPT Turing machine A: If A is given $\mathcal{H}$ and $y = \mathcal{H}(x)$, A can only with negligible probability return a $x'$ s.t. $y = \mathcal{H}(x')$.*

**Theorem 2 ($\pi_{\mathcal{P}\text{-M-SIG}}$ UC-realizes $\mathcal{F}_{\text{M-SIG}}$).** *Under assumptions 1, 2 and 3 and if $F_k$ and $\mathcal{E}^S$ are secure, $\pi_{\mathcal{P}\text{-M-SIG}}$ UC-realizes $\mathcal{F}_{\text{M-SIG}}$ with respect to adaptive adversaries in the proactive model, where the adversary may corrupt at most one of: M, T or S and D. Active corruption is allowed for all players except D, which can be passively corrupted.*

In the model we use here, $U$ stores his backup share of $d$. It could be argued that this is not realistic and we should extend the model with a separate player $D_U$ modeling whatever back-up storage $U$ is using. This is indeed possible, and our protocol would still be secure in this extended model against an adversary who may do corruptions as usual and in addition may corrupt $D_U$ passively, provided he never corrupts other players in a way that would give him both back-up shares (e.g., corrupting both $D$ and $D_U$ would be forbidden). We have omitted this for simplicity. Note that since we assume $U$ is never corrupted, we could in principle implement the back-up needed for *refreshment** by giving $U$ the complete key $d$. However, this solution is clearly dubious since it stores the entire secret key in a single location. Indeed, it is clearly insecure if we extend the model with $D_U$, which is why we prefer $\pi_{\mathcal{P}\text{-M-SIG}}$. Another advantage of $\pi_{\mathcal{P}\text{-M-SIG}}$ is that higher security during *refreshment** can be implemented by using two backup sharings. One sharing when $S$ has been corrupt, but is honest again, and an other sharing when $M$ has been corrupt, but is recovered. This improves security in a case where several adversaries work independently of each other, but each adversary corrupts at most one player. Such a case cannot covered by the standard model, where a single monolithic adversary is always assumed.

**Comments on the usage of Passwords** It should be emphasized that the refreshment protocol of *pwd* in Fig. 7 can run without the rest of refreshment. The reason for doing this is if the user suspects a corrupted terminal has been used. To ensure that none other than the current used terminal $T$ can change the password during *refreshment**, where a blank "old" password is used, a list of onetime passwords can be generated during key generation and sent to $U$ and stored by $D$.

## 6 Blinding Messages

A well-known technique by Chaum [8] can be used to blind the messages to be signed so that $S$ learns no information whatsoever on what is signed. The idea is that $M$, when it handles a message $m$, will choose a random $r \in Z_N^*$ and compute $b = \mathcal{H}(m)r^e \bmod N$, where $e$ is the public RSA exponent. Note that this can be feasible even for a computationally weak $M$ if a small public exponent is used. In the rest of the signature issuing, $\mathcal{H}(m)$ is replaced by $b$, and the PRF-value needed for outsourcing is computed from $b$. The blinding factor $r$ is sent to $T$. Since the signature issuing process is otherwise unchanged, it will eventually allow $T$ to compute $b^d \bmod N = \mathcal{H}(m)^d r \bmod N$, so by dividing out $r$, $T$ can recover the signature. On the other hand, $S$ has only seen $b$ which reveals nothing about $m$ since $r$ was uniformly chosen.

## 7 Implementing a prototype.

We have formulated a cryptographic approach to improve the security and mobility of the usage of digital signatures. If these improvements are to reach the

actual users in the real world, the cryptographic protocols has to be implemented in a way that makes it attractive for real users to use them. Therefore we are working together with experts in the field of "Human Computer Interaction" on a prototype that implements the described protocols. This is still work in progress, but the signature issuing phase of the protocol has been implemented. The implementation is done in Java using the Bouncy Castle library [16] for cryptographic tasks. We used a Sony Ericsson T850i cellphone to play the role of $M$ while $T$ was a standard PC, speaking to $M$ via Bluetooth, and to $S$ via SSL. We have verified that the signatures produced conform to the standard of the nation-wide PKI OCES[5] already in use in Denmark.

We observed that outsourcing the exponentiation from $M$ to $T$ gives a very significant speedup. With 1024 bit keys, exponentiation on the phone takes around 6 seconds. In a real use case, this makes the system seem heavy and slow and gives the impression that improved security degrades the user friendliness. Running the outsourcing protocol reduces the signature time to a fraction of a second and makes it seem as if the signing happens instantly. Of course, improved speed of exponentiation is possible by using native code instead of Java. But apart from the fact that portability would suffer if we do this, it is not clear that it would be enough: we would probably need to move to 2000 bit RSA in a real system which might cost up to a factor of 8 in performance.

## 8 Conclusion, Future work and Acknowledgement

We have proposed a model for personal digital signatures that we believe reflects reality better than previous proposals. We have proposed a protocol for signature generation that remains secure even if the computing equipment used is partially corrupt. Finally we have implemented the essential parts of the protocol and verified that it is practical. The protocol still assumes key generation as a trusted service, and in ongoing work we investigate methods for generating keys using a secure distributed computation, that takes into account the computational weakness of the mobile device.

We thank the anonymous PKC referees and Michael Steiner for comments that helped us improve the paper substantially.

## References

1. Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 593–611. Springer, 2006.
2. N. Asokan, Birgit Baum-Waidner, Torben P. Pedersen, Birgit Pfitzmann, Matthias Schunter, Michael Steiner, and Michael Waidner. In Gérard Lacoste, Birgit Pfitzmann, Michael Steiner, and Michael Waidner, editors, *SEMPER — Secure Electronic Marketplace for Europe*, volume 1854 of *Lecture Notes in Computer Science*, pages 45–64. Springer Verlag, August 2000. ISBN 3-540-67825-5.

[5] Danish for "Public Certificates for Electronic Services"

3. Dan Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2), 1999.
4. Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys. *J. ACM*, 48(4):702–722, 2001.
5. Peter Buhler, Thomas Eirich, Michael Waidner, and Michael Steiner. Secure password-based cipher suite for tls. In *NDSS*. The Internet Society, 2000.
6. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. (2005 version).
7. Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 404–421. Springer, 2005.
8. David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1982.
9. Ivan Damgård and Maciej Koprowski. Practical threshold RSA signatures without a trusted dealer. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2001.
10. Ivan Damgård and Gert Mikkelsen. On the theory and practice of personal digital signatures. In *Eprint Archive*, 2008.
11. Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Key-insulated public key cryptosystems. In Lars R. Knudsen, editor, *EUROCRYPT*, volume 2332 of *Lecture Notes in Computer Science*, pages 65–82. Springer, 2002.
12. Rosario Gennaro, Tal Rabin, Stanislaw Jarecki, and Hugo Krawczyk. Robust and efficient sharing of RSA functions. *J. Cryptology*, 13(2):273–300, 2000.
13. Amir Herzberg. Payments and banking with mobile personal devices. *CACM*, 46(5):53–58, 2003.
14. Gene Itkis and Leonid Reyzin. Sibir: Signer-base intrusion-resilient signatures. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2002.
15. Mohammad Mannan and Paul C. van Oorschot. Using a personal device to strengthen password authentication from an untrusted computer. In Sven Dietrich and Rachna Dhamija, editors, *Financial Cryptography*, volume 4886 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2007.
16. Legion of the Bouncy Castle. Bouncy castle crypto APIs, `http://www.bouncycastle.org`.
17. Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59, 1991.
18. Bryan Parno, Cynthia Kuo, and Adrian Perrig. Phoolproof phishing prevention. In Giovanni Di Crescenzo and Aviel D. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2006.
19. Tal Rabin. A simplified approach to threshold and proactive RSA. In Hugo Krawczyk, editor, *CRYPTO*, volume 1462 of *Lecture Notes in Computer Science*, pages 89–104. Springer, 1998.
20. Thomas Weigold, Thorsten Kramp, Reto Hermann, Frank Höring, Peter Buhler, and Michael Baentsch. The Zurich trusted information channel - an efficient defence against man-in-the-middle and malicious software attacks. In Peter Lipp, Ahmad-Reza Sadeghi, and Klaus-Michael Koch, editors, *TRUST*, volume 4968 of *Lecture Notes in Computer Science*, pages 75–91. Springer, 2008.