Protecting Electronic Signatures in Case of Key Leakage

Mirosław Kutyłowski[1](✉), Jacek Cichoń[1], Lucjan Hanzlik[1], Kamil Kluczniak[1], Xiaofeng Chen[2], and Jianfeng Wang[2]

[1] Faculty of Fundamental Problems of Technology, Wrocław University of Science and Technology, Wroclaw, Poland
{miroslaw.kutylowski,jacek.cichon,lucjan.hanzlik, kamil.kluczniak}@pwr.edu.pl

[2] State Key Laboratory of Integrated Service Networks (ISN), Xidian University, Xi'an, People's Republic of China
{xfchen,jfwang}@xidian.edu.cn

Abstract. We present a protection mechanism against forgery of electronic signatures with the original signing keys. It works for standard signatures based on discrete logarithm problem such as DSA. It requires only a slight modification of the signing device – an implementation of an additional hidden evidence functionality.

We assume that neither verification mechanism can be altered nor extra fields can be added to the signature (both as signed and unsigned fields). Therefore, the old software for signature verification can be used without any change. On the other hand, if a forged signature emerges, the signatory may prove its inconsistency with a probability close to 1.

Unlike fail-stop signatures, our method works not only against cryptanalytic attacks, but it is primarily designed for the case when the adversary gets the original signing key stored by the signing device of the user.

Unlike cliptographic constructions designed to defend against malicious implementations, we consider catastrophic situation when the key has been already compromised.

The technical idea we propose is an application of kleptography for good purposes. It is simple enough, efficient and almost self-evident to be ready for implementation of cryptographic smart cards of moderate storage and computational capabilities.

Unfortunately, we have also to bring into attention that our scheme has a dark side and it can be used for leaking the keys via the recent *subversion-resistant* signatures by A. Russell, Q. Tang, M. Yung and H.-Sh.Zhou.

This research has been supported by the Polish National Science Centre grant OPUS, no 2014/15/B/ST6/02837 and Polish-Chinese cooperation venture of Xidian University and Wrocław University of Science and Technology on Secure Data Outsourcing in Cloud Computing.

R.C.-W. Phan and M. Yung (Eds.): LNCS 10311, Mycrypt 2016, pp. 252–274, 2017.
DOI: 10.1007/978-3-319-61273-7_13

1 Introduction

In practice, undeniability of electronic signatures is based on the following assumptions:

1. Creation of a valid signature is possible only with the secret key corresponding to the public key used during the verification procedure,
2. The private key is stored only in a so called *signature creation device* which is under a sole control of the signatory,
3. The link between the public key and the signatory is reliably confirmed by means of a public key infrastructure.

Violation of any of these assumptions means that a signature cannot be trusted.

Note that the sole control condition has to indicate that no one but the signatory can activate the device for signing. This does not mean that the signatory has open access to the device memory, its secret keys etc. Indeed, such an access would mean that the signatory can export the key from the device and therefore we could not assume that the private signing key is stored on the device only.

Unfortunately, testing whether the above mentioned conditions are fulfilled might be far beyond the possibilities of a normal user entrusting the signatures:

1. Validity of the first condition depends on the state of the art of cryptanalysis. However, one cannot expect that a user can evaluate state-of-the-art of the research in this area and exclude possibility of a forgery. Note that we are not talking only about the academic research with publicly available results, but also about cyber war research.
2. The second condition is very hard to check, even for the signatory himself. First, the hardware manufacturer may include trapdoors enabling to reveal the private key stored in the signature creation device. There is a wide range of methods that can be applied for this purpose: kleptographic code (see e.g. [16] and its bibliography list), weaknesses of (pseudo)random number generators, hardware Trojans (see e.g. [1]), side channel information leakage (see e.g. [7]). Such trapdoors can be also created by simple implementation errors. However, even the signatory has no access to the internal state of the device, so such trapdoors cannot be detected by a direct device inspection.
   Certification and audit procedures aim to prevent existence of such trapdoors in certified products, but it has been pointed out that certification authorities might be coerced by state authorities to install certain trapdoors for the sake of national security. However, these trapdoors can be used without any control and even endanger public security.
   Some defense against these threats is possible – e.g. cliptographic techniques [19] aim to protect against malicious software implementations, including the key generation process.
3. Creating a reliable, efficient and cheap PKI infrastructure is a problem itself. Last not least, one has to be sure that the most advanced players cannot create rogue certificates. From the past experience (see [12]), we know that this cannot be completely excluded.

In this paper we focus on the second condition. One line of R&D work is to provide secure hardware so that there are neither trapdoors nor malicious ways to extract the signing secret keys against the will of the signatory and the manufacturer. However, it is extremely hard to evaluate effectiveness of the current solutions for the sheer reason that the most dangerous and powerful adversaries may treat their capabilities as top secrets.

Key Generation Process. One of the Achilles heels of the signature schemes currently implemented on the smart cards is secret key generation. There are a few options, each of them involves critical threats:

1. **Key generated by a service provider and installed on the signature creation device:** one cannot prove that the service provider does not retain a copy of the key for malicious purposes. If they are later used only in a few cases, then this does not even endanger the reputation of the service as forgery might be hard to prove.
   Unfortunately, the current legal regulations do not ease the situation: e.g. the recent eIDAS framework of European Union [14] admits creation and keeping backup copies of secret signing keys as a service. It is required that *appropriate security level is assured*, however it does not mean that a rogue service provider cannot break these rules. Even worse, the current state-of-the-art of cryptographic research provides tools to erase all traces of misbehavior.
2. **Key generated by the user and installed on the signature creation device:** this option is frequently forbidden by legal rules: the device must not enable existence of the secret key outside of it. The only exception is generating key by a trusted service provider as mentioned by the first option. Note that admitting the user to install the key creates plenty of opportunities to steal this key by rogue software on the computer used to generate and upload the key. The user is likely to be unaware of this fact.
3. **Key generated itself by the signature creation device:** the procedure to create cryptographic keys on the signing device might be a fake one. A rogue device might "generate" the key that already has been stored on the device or is predictable by the manufacturer. As in response the device owner obtains only a public key, it is infeasible to detect such misbehavior.

We may conclude that the provider of the signature creation devices has quite realistic ways to gain access to the signing keys contained in these devices and thereby has a real opportunity to forge signatures.

Of course, there are some relatively simple solutions that would make such forgeries much harder or even impossible. For example, a multi-party key generation protocol executed by the signature creation device and another independent device of this user would help a lot (by "independent" we mean in particular "coming from a different manufacturer that cannot be coerced by the same authority"). Also using two different signature creation devices and a certificate stating that a signature is valid only when co-signed by the second device would provide more security in the practical sense.

Unfortunately, these simple solutions are not really pragmatic. Our experience is that in practice all solutions that require major changes in the already deployed systems have no or very low chances to be accepted by the industry and decision makers. As a reaction to such proposals we expect rather arguments to ignore "theoretical and purely academic threats not justified by a realistic risk analysis". If a new method is fully compatible with the already available and deployed products, then the chances to be accepted increase a lot.

Our Goal. We aim to create security mechanisms against forging signatures by an adversary having access to the private signing keys. Such an adversary can create signatures that cannot be distinguished by the standard means from the signatures created by the private key owner. As it seems to be hard to change the verification process already described in the standards, our goal is to

– detect signatures created with stolen signing keys,
– create an evidence that can be used to convince a judge about the forgery without invalidating the remaining signatures.

We aim to address the situation where:

– the manufacturer provides a device with a signing key already installed, or
– the key generation procedure is executed by the (black box) device, but we have no insight into this process, or
– some components are ill implemented (e.g. the (pseudo)random number generator has low entropy output), and therefore the signing key is weak by design.

For this purpose we enable the user to install an auxiliary parameter on the signature creation device after he gets the device to his hands. An important feature is that this parameter is not known to the manufacturer or the party delivering the device. As the signing device is usually supposed only to create signatures, it might be hard in practice to create a secret channel to the manufacturer that would enable to leak the signing key.

A naïve way to implement this idea would be to use an extra secret as a seed for a PRNG creating random exponent k i.e. in case of DSA signatures. Revealing the seed would immediately enable to distinguish the forged signatures from the signatures created by the card. Unfortunately, the user himself would be able to derive the secret key installed by the device, which contradicts the basic requirements e.g. from [14]. Indeed, in case of the signature creation devices the legitimate user of the device should be considered as an adversary, since in practice he might be tempted to find an evidence that his device has been tampered and therefore to claim that some of his past signatures have been forged.

One might hope that fail-stop signatures [11] solve the stated problem. Unfortunately, this is not true, as fail-stop signatures protect solely against cryptanalytic attacks. They are useless against an adversary holding the original signing keys.

Assumptions. We aim to provide a solution that would be relatively easy to deploy. Therefore we assume that:

1. The current signature standards should be used in an essentially unchanged form; only details of implementation should provide additional features for the device holder. Nothing should change for a verifier of electronic signatures.
2. The solution does not require any additional interaction during device delivery from the manufacturer to the end user. We are focused on the situation that the signature creation device is delivered ready to use with the signing key and a certificate for this key already preinstalled.
3. The solution should be implementable on devices such as smart cards with moderate resources (computing speed and memory size). These devices are assumed to be tamper proof (if they are not, then they cannot be used as signature creation devices for the general reasons).
4. For the signatory, creating a signature should be as convenient as in case of the standard signature creation devices.
5. In case of a forgery with original keys, detection probability should be high enough to discourage such behavior.

Example Application Scenario. A typical application case we have in mind is the following. A financial institution or an organization of financial institutions issues electronic ID cards for their customers. The ID cards enable creation of electronic signatures used to authenticate documents provided by the customers. In order to ease the card usage and simplify the logistics, the ID cards are delivered ready to use. As the card issuer holds the most relevant data about the customer enabling him to monitor card usage, it does not make sense to delegate the cards management chores (such as keeping revocation lists) to a third party Certificate Authority. However, if there is a dispute between the customer and the financial institution, the customer may challenge the signatures created by himself and claim that the institution has retained his signing key and used it to forge these documents. This is a crucial issue and the solution proposed in this paper aims to prevent such a situation.

Our Contribution and Paper Overview. We present a generic solution for randomized signature schemes based on the Discrete Logarithm Problem, including in particular ElGamal, DSA, ECDSA, and Schnorr signatures. The mechanism is based on a pair of hidden keys that enable to fish out the signatures created by the legitimate signature creation device (this device stores the public control key for signature creation). Our solution enables separation of trust: the hidden keys might be uploaded by the owner of the device.

In Sect. 2 we present our generic construction. In Sect. 3 we provide arguments showing that the modified schemes are secure just as the underlying signature schemes. Section 4 shows a dark side of the scheme – usage for leaking the key without using random numbers generated on the smart card. In Sect. 5.1 we discuss the implementation problems and in particular time complexity for signature creation on a smart card. In Sect. 5.2 we present and discuss a proof of concept implementation. Section 6 is devoted to the related previous work.

2 Scheme Description

2.1 Outline of the Solution

In our scheme we redesign the life cycle of secure signature creation devices. The following parties are involved:

Manufacturer: he creates a signature creation device - both the hardware and the software installed there. In particular, he either personalizes the device and installs there the private signing keys on behalf of the final user, or installs software responsible for key generation by the device.

Signatory: the person that holds the signature creation device and controls the physical access to it after getting it from the manufacturer. Apart from the signing activities, he is involved in interactions with his signature creation device aiming to protect against an adversary, who may potentially gain access to his signing key. In particular, the signatory holds an additional private *hidden control key* assigned to the signature creation device. He initializes the device by installing the corresponding hidden public key on his signing device. He keeps this key confidential and shows it to nobody but to the judge[1].

In case of a fraud with the original signing keys, the signatory interacts with the judge and proves forgery.

Verifier: a recipient of digitally signed documents. He runs the standard signature verification procedure to check validity of a signature.

Adversary: a party attempting to create a signature that would be attributed to the device of the attacked signatory. After delivering the device to the signatory, the adversary cannot directly interact with the device, but we assume that he holds the signing key stored in this device. He may also see the signatures created by the device and even request signatures under messages of his choice.

Judge: is a party that resolves disputes about claimed signature forgeries. For this purpose he interacts with the signatory.

In fact, unlike in the standard cryptographic literature, the signature creation device should be regarded as a scheme participant. In particular, it interacts with the signatory in a strictly defined way. So in particular, it should be tamper resistant. Its life cycle consists of the following phases:

Phase 1: device creation and delivery. The signature creation device is produced and delivered to the signatory. We assume that its whole memory contents at that time is known to the adversary and that the device might be personalized for the signatory.

Phase 2: hidden keys initialization. A fresh signature creation device gets the hidden public key installed by the signatory. (In Sect. 6, we indicate that it is possible to change the hidden keys, however this cannot be done completely freely.) The signatory does not certify the hidden public key, however

[1] Or even not to the judge, if he is only semi-trusted.

it might be helpful to register the signature creation device. According to our scheme, for this purpose he deposits a number of signatures created by the so initialized device.

Phase 3: regular usage. During this phase the signatory uses the device and creates digitally signed documents. At the same time the adversary may forge signatures using for this purpose the original signing keys of the signatory. Such forged signatures are used by the adversary for presumably malicious purposes.

Phase 4: forgery detection and proof. This phase may occur, when the signatory becomes aware of valid signatures that are attributed to him, but which have not been created by using his signature creation device. In this case the signatory interacts with a judge. (The adversary is not involved in this phase, since in general neither the signatory nor the judge can indicate who is the adversary.) Based on the undeniable cryptographic proof the judge can declare the challenged signature(s) as not created by the device of the signatory and therefore attributed to an unknown adversary.

Phase 5: termination of use. In this phase device usage is terminated; possibly all already existing signatures or some of them get revoked. This phase normally follows Phase 4, when the judge decides that the presented signature has been forged. In a standard situation, termination is due to aging of the signature creation device.

2.2 Preliminaries

First let us discuss a few components used by the proposed scheme.

Signature schemes involved: Our method will apply to signature schemes based on Discrete Logarithm Problem for a cyclic group $\mathcal{G}$ of a prime order q. For $\mathcal{G}$ we shall use the multiplicative notation, however the scheme can be applied for elliptic curve signatures in exactly the same way. The solution works for signature schemes such as ElGamal, DSA, Schnorr, where one of the signature parameters available for the verifier is an element $r = g^k$, where k is created at random.

Hidden control keys: The main feature of signature creation devices executing our scheme is that they contain an extra public key $V = g^v \in \mathcal{G}$. Each signature creation device implementing the scheme must have its own key V installed after handling the device to the signatory. The link between the device and V should be verifiable by a judge. Moreover, the hidden private key v should be known to the signatory, whose responsibility is to prove in the court that certain signatures have been forged, if this unfortunate case occurs. The hidden key v should not be installed on the signing device.

Data signed: the signature schemes concerned apply a hash function together with the core signing operations. According to the common practice, legal requirements and industrial standards, the hash function is applied not to a raw document M, but to M encoded together with some meta-data. The meta-data contain in particular the time of creating the signature (see for instance the popular XAdES standard and its QualifyingProperties: SignedProperties: SigningTime). Following RFC 2985 [9] we assume that the signing time is given according to the ISO/IEC 9594-8 standard UTC Time, and according to RFC 2630 [6] MUST include seconds. The following issues are important for our scheme:

- For a signing time T, we utilize the field denoting the seconds, say t. For simplicity we assume that $0 \leq t \leq 59$ and ignore the rare cases when a minute has 61 s.
- The signature creation device can adjust a signing time by causing a delay of, say, two seconds.
- Before the secret key is used by the signing procedure, a small preprocessing takes place, allowing to adjust the parameter r to the signing time in a hidden way.

Truncated hash function TruncHash: Our scheme uses a hash function

$$\text{TruncHash} : \mathcal{G} \rightarrow \{0, \ldots, 59\}$$

For the sake of security proof we have to assume that TruncHash is a pseudorandom function that can be modelled by a random oracle.
Note that for TruncHash we do not aim to achieve collision freeness or any property of this kind. What we really need is that a fair advantage in guessing the value of $\text{TruncHash}(x)$ indicates that its argument x has been presented first.
For the sake of a practical implementation, TruncHash could be taken as a truncated value of a hash function implemented on the smart card (e.g. SHA-256).[2]

2.3 Signing Scheme Procedures

In order to focus attention of the reader, we describe a scheme based on Schnorr signatures. We also skip most of the details that do not differ our solution from the standard one. Let $\mathcal{S}ignDev$ stand for a signing device.

I. Generating a key pair for a user. We do not alter the way of generating the signing keys. We assume that after this phase:

- $\mathcal{S}ignDev$ contains a private signing key $x < q$ chosen at random,
- the public key $Y = g^x \in \mathcal{G}$ has been exported outside $\mathcal{S}ignDev$,
- $\mathcal{S}ignDev$ is in the state requiring installing the hidden control keys.

II. Installing the hidden control keys. This procedure is executed by the signatory interacting with his device $\mathcal{S}ignDev$ with already instantiated private signing key x.

[2] There are some low level implementation issues, since hash function is typically hardware supported, but truncation presumably cannot be executed on the cryptographic co-processor and therefore might be relatively slow.

1. The signatory chooses the hidden secret key v, $v < q$, at random and computes $V := g^v \in \mathcal{G}$.
2. The signatory uploads V to $\mathcal{S}ignDev$[3]. The device $\mathcal{S}ignDev$ stores V as its hidden key.
3. $\mathcal{S}ignDev$ enters the state in which it can be used only for creating signatures.
4. The signatory creates a few signatures (as described below) and deposits them by a third trusted party.

Remark 1. In practice, the signatory uses another (independent) device (presumably his PC) to create v and V. Insecurity of this auxiliary device may challenge the effectiveness of the forgery detection mechanism, however it does not endanger the underlying signature scheme. Involving a PC does not create an additional risk, as an adversary that controls it may replace the documents to be signed by the documents of his choice at the moment when the signatory authorizes $\mathcal{S}ignDev$ to create a signature.

III. SIGNING PROCEDURE. Creating a signature consists of two phases. The first phase can be executed in advance before the data to be signed (or its hash) are transmitted to the signature creation device. The second phase corresponds to the standard signing procedure.

First, for the reader's convenience we recall the Schnorr signing algorithm for a signing key x and a message M (note that M is the original document appended with the signature's meta-data, including in particular the signing time):

CREATION OF SCHNORR SIGNATURES

```
1. choose $k$ at random
2. $r := g^k$
3. $e := \text{Hash}(M||r)$
4. $s := (k + x \cdot e) \bmod q$
5. output $(s, e)$ as a signature of $M$.
```

CREATION OF SCHNORR SIGNATURES - FORGERY EVIDENT VERSION

This procedure is executed by the device $\mathcal{S}ignDev$ in interaction with (the computer of) the signatory. (Of course, a signing request has to be authorized by the signatory – providing the correct PIN is a minimum requirement.)

Phase 1 (preprocessing):

```
1. create an empty array $A[0 \ldots 59]$
2. choose $k$ at random
3. $U := V^k$
4. $i := 0$
5. repeat $\Delta$ times:            // $\Delta$ is a constant discussed later
   5.1. $z := \text{TruncHash}(U, M)$
```

[3] In order to secure the logistic chain, the smart card may require presenting a one-time initialization PIN by the signatory.

```
    5.2. $A[z] := i$
    5.3. $i := i + 1$
    5.4. $U := U \cdot V$
```

After Phase 1 the array A and the value k are retained for Phase 2.

Phase 2 (the main signing part):

```
1. let $T$ be the signing time (in the UTC format), let $t$ be the value of the seconds field of $T$
2. wait until $A[t]$ is nonempty
3. $r := g^{k+A[t]}$
4. having $r$ already computed (see e.g. the first two steps of the Schnorr algorithm), proceed with the signature creation algorithm for the message to be signed appended with meta-data containing the signing time $T$.[4]
```

Remark 2. The choice of the parameter Δ will be discussed in Sect. 5.1. It must be large enough to fill majority of positions in the array A and thereby reduce the waiting time in the second step of Phase 2.

Remark 3. Note that if $A[t]$ is nonempty and contains $j < \Delta$, then the entry j has been inserted for $U = V^{k+j}$ and $t = \text{TruncHash}(U, M)$. Of course, during the first phase $A[t]$ might be overwritten many times, but it does not change the just mentioned property.

IV. Verification procedure. The verification procedure is unchanged and executed according to the standard version of the underlying signature.

Remark 4. Note that our signature creation process yields signatures in exactly the same format as in case of the underlying scheme. So, we do not need any modification of the verification software. In fact, we shall show that the verifier cannot recognize whether the signing procedure has been modified.

V. Forgery detection procedure. The test concerning a signature of M with the creation time T (as stated in the signed meta-data) is as follows:

1. reconstruct the value r, e.g. for a Schnorr signature (s, e), compute $r := g^s/Y^e$
2. check
$$\text{TruncHash}(r^v, M) \overset{?}{=} t \qquad (1)$$
where t denotes the value of the seconds field in T.
3. output `forgery`, if the equality (1) does not hold.

Remark 5. The detection procedure requires knowledge of v – the hidden control key. It must be kept secret by the signatory. Therefore there are two procedures: one concerning forgery detection and the second concerning proving forgery against a judge.

[4] In the cryptographic standards the signing time T is treated as meta-data and not a part of the message to be signed, in the cryptographic literature there is no such distinction and the message is understood as a message together with the meta-data.

Remark 6. There is a possibility of a false positive result – by pure luck the adversary creating a signature may use r such that equality (1) holds. This happens with probability $\approx \frac{1}{60}$ for each single signature. This probability is high from the point of view of cryptography, however in this case we are talking about disclosure of malicious behavior of the system provider, who gets into criminal charges with probability $\frac{59}{60}$, if he dares to use a stolen key even once! Note also that we are not talking about general probability to forge a signature, but about the probability to use a forged signature, when the private signing key has already been exposed to the adversary.

VI. Forgery proof procedure. This procedure is executed by the signatory and the judge. First we describe the general framework and later describe the details of the core Proof Procedure:

1. The signatory presents the judge the hidden public key V.
2. The signatory presents the judge the registered signatures created right after installing the hidden control keys.
3. The judge and the signatory run interactively the Proof Procedure for these signatures and V. If the result is negative, then the signatory's claim gets rejected and the procedure terminates.
4. The signatory presents the judge the alleged forged signature $\mathcal{S}$ with the signature creation time T.
5. The judge and the signatory run the Proof Procedure for $\mathcal{S}$. If the result is negative, then the signatory's claim gets rejected and the procedure terminates. Otherwise the judge declares $\mathcal{S}$ as forged.

Proof Procedure

Consider a signature $\mathcal{S}$ of a message M with a signing time T, with t denoting the seconds field, for the hidden public key V and the hidden private key v:

1. The judge and the signatory reconstruct the value r from the signature creation process of $\mathcal{S}$ – just as in case of a regular signature verification.
2. The signatory computes $u := r^v$ and presents u to the judge.
3. The judge rejects the forgery claim if $\mathrm{TruncHash}(u, M) = t$.
4. The signatory and the judge perform an interactive zero knowledge proof of equality of discrete logarithms for the pairs (g, V) and (r, u). For instance, one may run a number of times the following standard procedure:
   (a) the signatory chooses σ at random and presents
   $$v_1 = g^{v\sigma}, v_2 = r^{v\sigma}$$
   (b) the judge chooses a bit b at random,
   (c) if $b = 0$, then the signatory reveals σ and the judge checks that $v_1 = V^\sigma$, $v_2 = u^\sigma$,
   (d) if $b = 1$, then the signatory reveals $\delta = v\sigma$ and the judge checks that $v_1 = g^\delta, v_2 = r^\delta$.
5. If the equality of discrete logarithms test fails, then the judge rejects the forgery claim.

Remark 7. The above procedure can be modified so that V is not presented to the judge (this approach should be applied, if the judge is only semi-trusted.) In this case the signatory does not create a proof of equality of discrete logarithms for (g, V) and (r, u) as described above, but for (r_0, u_0),..., (r_m, u_m) and (r, u), where (r_0, u_0),..., (r_m, u_m) come from the signatures registered right after the personalization of the $\mathcal{S}ignDev$ device with the hidden keys or from the signatures created in front of the judge with $\mathcal{S}ignDev$.

Remark 8. One may speculate that the signatory might himself break into his signing device $\mathcal{S}ignDev$. Then having the secret signing key and the hidden key, the signatory would be able to create signatures that would be recognized as forged by the judge. However, according to the manufacturer's declaration $\mathcal{S}ignDev$ is tamper resistant.

3 Security of the Scheme

3.1 Resilience to Forgeries

Potentially, the special properties of the signatures created according to the scheme proposed in Sect. 2 might ease forging signatures by the third parties. Namely, we can consider the following game:

Game A

1. The adversary may request signatures of the messages of his choice at time of his choice. They have to be created by a device implementing the algorithm from Sect. 2.
2. The adversary presents a message and its signature S.

The adversary wins the game if S has not been created during step 1 and the standard verification of S yields a positive result.

Note that we do not demand that S would pass the forgery detection procedure. Moreover, the signature S may concern a message that has already been signed during Step 1 (re-signing an old message).

We can also define an analogous game for standard signature creation:

Game B

1. The adversary may request signatures of the messages of his choice at time of his choice.
2. The adversary presents a message and its signature S.

The adversary wins the game if S has not been created during step 1 and the standard verification of S yields a positive result.

Assume that an adversary $\mathcal{A}$ can win Game A. We use it to win Game B. Namely, we choose v at random and set $V = g^v$. Then we create the input for Game A. If the adversary of Game A asks for a signature over M at time T with t being the number of seconds, we use the oracle from Game B. When it returns a signature $(r, \ldots)$, then we compute $\text{TruncHash}(r^v, M)$. If it equals t, then we pass the signature $(r, \ldots)$ to Game A. If not, then we repeat the request for the oracle of Game B, until this property is fulfilled. Since there are only 60 possible values of TruncHash, after a short time we get a signature that can be passed to Game A.

Obviously, the signature returned by the adversary $\mathcal{A}$ from Game A can be used as a response of the adversary in Game B. Thereby we get the following result:

Theorem 1. *If an adversary can win Game A (forge a signature based on a collection of signatures created according to the mechanism from Sect.* 2), then an adversary can win Game B (forge a signature in the regular case).

Remark 9. Note that the opposite direction is not immediate and may require extra assumptions: in the attack for Game A we create signatures that come from a subset of the set of all signatures. In theory, a successful attack from Game B could fail, if we are limited to such a subset.

3.2 Indistinguishability

Our second goal is to show that an adversary not knowing the control key $V = g^v$ cannot distinguish between the signatures passing the forgery detection test and signatures that would be found as forged. (This notion is basically the same as *indistinguishability against Subversion Attacks* from [18].) Namely, we consider the following game:

Game C

1. The adversary may request signatures of the messages of his choice at time of his choice. For each signature $(r_i, \ldots)$ over M_i, he gets additionally z_i, where
$$z_i = \text{TruncHash}(r_i^v, M_i)$$
2. The challenger chooses a bit b at random.
3. The challenger presents a signature $(r, \ldots)$ over M that has not been created during the first stage and a value z, which equals $\text{TruncHash}(r^v, M)$, if $b = 0$, and a random value $z \neq \text{TruncHash}(r^v, M)$, if $b = 1$.
4. The adversary returns a bit $\bar{b}$.

The adversary wins the game if $b = \bar{b}$.

In our proof we refer to Correlated-Input Secure Hash Function introduced in [5]. This model seems to better reflect the required properties than the Random Oracle Model and addresses directly the threats occurring in practice - including in particular our case. Moreover, for our purposes it suffices to use only a limited version of Correlated-Input Hash Function Assumption. Namely, we consider the following game:

Correlated TruncHash Values Game

```
choose pairwise different elements $k_1, \ldots, k_n \leq q$
    (an arbitrary strategy may be applied)
choose $M_1, \ldots, M_n \in \mathcal{G}$
    (an arbitrary strategy may be applied)
```

```
choose $V$ at random
$h_i :=$ TruncHash$(V^{k_i}, M_i)$ for $i = 1$ to $n$,
choose $M$ and $k \neq k_1, \ldots, k_n$
$h^{(0)}_{n+1} :=$ TruncHash$(V^k, M)$
choose $h^{(1)}_{n+1} \in \{0, \ldots, 59\} \setminus \{h^{(0)}_{n+1}\}$ at random
choose $b \in \{0, 1\}$ at random
$\hat{b} := \mathcal{A}(k_1, \ldots, k_n, k, M_1, \ldots, M_n, M, h_1, \ldots, h_n, h^{(b)}_{n+1})$.
```

$\mathcal{A}$ wins the game if $b = \hat{b}$. The advantage of $\mathcal{A}$ is defined as $p - \frac{1}{2}|$ where p is the probability to win the game by $\mathcal{A}$.

Note that in the above game the only value not available to $\mathcal{A}$ is V. We know exactly the relationship between the first arguments of TruncHash due to the knowledge of the exponents $k_1, \ldots, k_n, k$. However, the hash function TruncHash hides the arguments used, so we cannot derive V^k. The game describes the chances of the adversary to deduce the control value for r, given the correct values for some other $r_1, \ldots, r_n$.

Assumption 2. (Reduced version of Correlated-Input Hash Function Assumption). The advantage of the adversary $\mathcal{A}$ in the Correlated Hash Values Game is negligible.

Note that the adversary can win the Correlated TruncHash Values Game, if he can win Game C. Indeed, the adversary may choose at random a signing key x. Then, knowing the exponents $k_1, \ldots, k_n, k$ he may create the corresponding signatures for $M_1, \ldots, M_n, M$. Therefore we may immediately conclude as follows:

Theorem 3. *Each adversary has a negligible advantage in Game C, if the (reduced) Correlated-Input Hash Function Assumption holds for the function TruncHash.*

4 The Dark Side of the Scheme

In this section we remark that the core mechanism of the scheme presented in Sect. 2 can be used for evil purposes as well. Jumping ahead to the related work (Sect. 6), let us recall that there have been attempts to eliminate software subversion attacks on signature schemes [18,19]. Their general recommendation is to use deterministic signature schemes and to involve in certain way a hashing function during key generation process. In this way we defend ourselves against choosing the keys that are advantageous for the adversary.

Unfortunately, the proposed approach does not fully implement the concept "nothing up my sleeve" and the security claims from [19] turn out to be incomplete. Let us sketch shortly how to leak the key with their scheme:

– There is no room for randomness during execution of the protocol, however the user cannot fully control the time of signature creation. A malicious software can postpone signature creation by, say, 1 or 2 s. This cannot be really

observed by the user and can be attributed to many hardware issues. (For instance, a smart card may suffer from communication errors and attribute the delays to them).
– The time delay may be aimed to create a subliminal channel: the device tries to adjust some number of bits to leak a key. Say, it takes the signature s (created honestly according to the protocol) and computes $D := \mathrm{Hash}(S, K)$, where K is a trapdoor key. Then it takes, say, the last 8 bits of D and treats them as an 8-bit address a. If the 9-th bit of S is the same as the ath bit of the 256-bit signing key, then the signature S is called *good*.
– The signing procedure is adjusted as follows:
  (1) the device creates a signature S taking into account the current time T. If S is *good*, then it releases S. Otherwise it waits one second and goes to step 2.
  (2) the device creates the next signature S' and releases it no matter whether it is *good* or not.
– The adversary collects a number of signatures created by the device. Each signature indicates the value of one bit of the signing key – but of course only the values coming from *good* signatures are true. Therefore for each key position the adversary gathers the values indicated by the signatures. As at average 75% of all signatures are *good*, with enough signatures it is possible to recover the signing key taking into account relevant statistics and enhancing them by brute force at positions where the statistics do not deliver a firm answer.

Note that if the signature creation device has to implement the scheme presented in Sect. 2, then installing the key leaking procedure described above becomes more problematic. Indeed, leaking the key bits requires adjusting the signing time and thereby creates some delays. However, this must be implemented on top of the procedure described in Sect. 2 which already causes some delays. So together the delay might be too high and easily observable.

5 Implementation Issues

When electronic signatures for real world applications are concerned, we have to take care about practical feasibility of the proposed solution. For instance, a protocol using non-standard cryptographic operations not implemented on the smart cards available on the market has almost no chance to be deployed in practice due to high costs of redesign of the card's cryptographic coprocessors.

For this reason we discuss the problems of a time delay introduced by the scheme from Sect. 2. This is a crucial issue, since there is a strict limit for a signature creation process – it should not exceed 2–3 s. Otherwise, the users get annoyed by the long processing time and generally do not accept the solution. (Interestingly, reducing the processing time does not make sense either, since the user should observe that the smart card is performing some computations.)

In order to check computational complexity in reality, we have also implemented our solution on MULTOS smart cards (the cards where the operating system enables access to low level operations for the programmer). The results show that no technical problems emerge for our scheme.

5.1 Empty Places in Array A

Creating a signature according to the procedure described in Sect. 2 at time t fails, if $A[t]$ is empty. In this case, the signature creation process gets postponed 1 s and the next attempt occurs at time $t+1$.

From the mathematical point of view we have the following problem. There are n bins and k balls. Each ball is placed in one of the bins, the target bin is chosen uniformly at random, independently of other balls. We define the event $\mathcal{E}_i^{(n,k)}$ meaning that the bin $i \bmod n$ is empty after inserting all balls. Then let $p_{n,k,a}$ denote the probability

$$\Pr[\mathcal{E}_{i+1}^{(n,k)} \wedge \mathcal{E}_{i+2}^{(n,k)} \wedge \ldots \wedge \mathcal{E}_{i+a}^{(n,k)}] = \left(1 - \tfrac{a}{n}\right)^k$$

The probability of hitting a non-empty position in the first trial is therefore $\Pr[\neg\mathcal{E}_i^{(n,k)}] = 1 - (1 - \frac{1}{n})^k$. For $n = 60$ and $k = 120$ this equals approximately $1 - 1/e^2 \approx 0.864665$. The concrete values for $p_{60,120,a}$ are as follows:

a	1	2	3	4	5
$p_{60,120,a}$	0.133	$1.7 \cdot 10^{-2}$	$2.1 \cdot 10^{-3}$	$2.5 \cdot 10^{-4}$	$2.92 \cdot 10^{-5}$

Let us also consider

$$L_{i,n,k} = \min\left\{a : \neg\left(\mathcal{E}_{i+1}^{(n,k)} \wedge \mathcal{E}_{i+2}^{(n,k)} \wedge \ldots \wedge \mathcal{E}_{i+a}^{(n,k)}\right)\right\}$$

That is, $L_{i,n,k}$ corresponds to the number of steps required to find a nonempty position in the array A starting at position $i+1$. The probability distribution of $L_{i,n,k}$ does not depend on i, so we will write $L_{n,k}$ instead of $L_{i,n,k}$. Then

$$\Pr[L_{n,k} > a] = \Pr[\mathcal{E}_1^{(n,k)} \wedge \mathcal{E}_2^{(n,k)} \wedge \ldots \wedge \mathcal{E}_a^{(n,k)}]$$

Then the expected value of $L_{n,k}$ can be computed as follows:

$$\begin{aligned} \mathrm{E}[L_{n,k}] &= 1 + \textstyle\sum_{a=1}^{n} \Pr[L_{n,k} > a] = 1 + \sum_{a=1}^{n-1} \left(1 - \frac{a}{n}\right)^k \\ &= 1 + \tfrac{1}{n^k} \textstyle\sum_{a=1}^{n-1} (n-a)^k = 1 + \frac{1}{n^k} \sum_{a=1}^{n-1} a^k \end{aligned}$$

Obviously, if $k_1 < k_2$, then $\mathrm{E}[L_{n,k_1}] > \mathrm{E}[L_{n,k_2}]$. Moreover, $\lim_{k\to\infty} \mathrm{E}[L_{n,k}] = 1$. The expression $1 + \frac{60}{k+1}$ is a quite good upper approximation of $\mathrm{E}[L_{60,k}]$ (see Fig. 1).

For $n = 60$ and $k = 1, \ldots, 240$ we may derive the following concrete values:

k	10	60	120	240	500
$\mathrm{E}[L_{60,k}]$	5.96..	1.56..	1.15..	1.01..	1.00022..

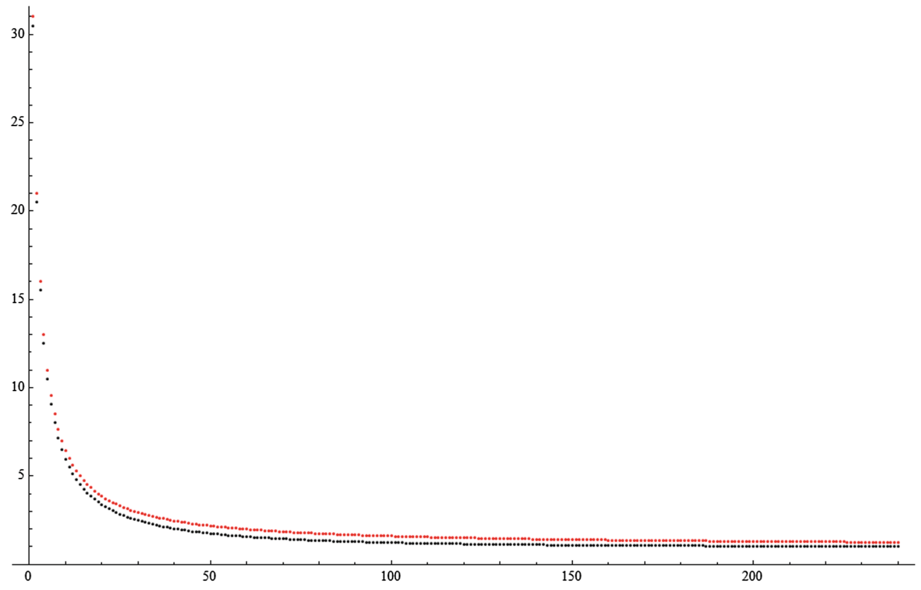

Fig. 1. The expected value $\mathrm{E}[L_{60,k}]$ (black points) and its approximation by $1 + \frac{60}{k+1}$ (red points). The value of k is on the x-axis. (Color figure online)

In particular it means that for $k = 120$ the array A is almost filled and the expected position of the first nonempty entry is in most cases the starting position.

We have executed a number of trial executions of the preprocessing stage (implemented in Python) for $\Delta = 100$. We have ran three series of 50 executions of the preprocessing stage and we have counted the number of blocks of empty entries of a given length in the array A. (For this purpose we assume that the entries $A[59], A[0]$ are the neighboring ones.)

- In the first series of 50 executions we have observed:
  - 360 blocks of length 1 (i.e. empty entries preceded and followed by non-empty entries),
  - 64 blocks of 2 empty entries,
  - 10 blocks of 3 empty entries,
  - 1 block of 4 empty entries,
  - 2 blocks of 5 empty entries.

  The average number of empty blocks in a single execution therefore equals 7.2 blocks of length 1, 1.28 blocks of length 2, 0.2 blocks of length 3, 0.02 blocks of length 4, 0.04 blocks of length 5.
- In the second series of 50 executions we have observed: 375, 68, 17, 3 empty blocks of the length, respectively, 1, 2, 3, 4.
- In the third series of 50 executions we have observed: 376, 75, 11, none, 2 empty blocks of the length, respectively, 1, 2, 3, 4, 5.

5.2 MULTOS Trial Implementation

In order to estimate complexity of the proposed scheme in a real environment we have implemented the most critical part – the preprocessing phase – on the standard MULTOS smart cards. Note that this is the only potential bottleneck of the scheme. Indeed, the verification procedure is exactly the same as in the standard case, while forgery detection and proof are executed only occasionally (anyway, they are quite efficient). We have used the 2048-bit MODP group with a 224-bit prime order subgroup defined in RFC 5114. Below we include the source code of this implementation.

Note that this implementation uses only the high level API of the MULTOS standard. So in some sense we perform the worst case experiment, as we use modular arithmetic instead of more efficient elliptic curves. This influences both the time complexity of standard operations used for signature generation, as well as auxiliary operations such as an implementation of TruncHash (in the code presented below this is SHA-1 followed by reduction modulo 60). Of course, a low level implementation should be much more efficient.

```
#include"main.h"

#define FIELD_LEN 256
#define MESS_LEN 256
#define EXP_LEN 28
#define ARR_LEN 60

#pragma melpublic
BYTE tmp[FIELD_LEN+MESS_LEN];
BYTE k[EXP_LEN];
BYTE zb[20];
BYTE kk[8];
struct
{
  BYTE buffer[ARR_LEN];
} apdu;

#pragma melstatic
BYTE p[] = {0xAD,0x10,0x7E,0x1E,0x91,...};
BYTE g[] = {0xAC,0x40,0x32,0xEF,0x4F,...};
BYTE V[] = {0x88,0x31,0x1a,0xf3,0x91,..};
BYTE v[] = {0x80,0x1C,0x0D,0x34,0xC5,..};
BYTE mess[] = {0x01,0x02,0x03,0x04,0x05,..};

void main(void) {
int i=0,j=0;
int delta = 100;
DWORD z;
BYTE t;

  multosGetRandomNumber(kk);
  memcpy(k,kk,8);
  multosGetRandomNumber(kk);
  memcpy(k+8,kk,8);
  multosGetRandomNumber(kk);
  memcpy(k+16,kk,8);
  multosGetRandomNumber(kk);
  memcpy(k+24,kk,4);
  multosModularExponentiation(EXP_LEN,FIELD_LEN,
  k,p,V,tmp);
  memcpy(tmp+FIELD_LEN,mess,MESS_LEN);
  for(i=0;i<delta;i++){
  multosSHA1(FIELD_LEN+MESS_LEN,tmp,zb);
  t = zb[0];
  __push(t);
  t = zb[1];
  __push(t);
  t = zb[2];
  __push(t);
  t = zb[3];
  __push(t);
  __code (__STORE, &z, 4);
  z = z
  apdu.buffer[z] = i;
  multosModularMultiplication(FIELD_LEN,p,tmp,V);
  }
multosExitLa(ARR_LEN);
}
```

The detailed results regarding the time required to execute Phase 1 for the above code on MULTOS are given in Table 1. Note that the time lower than 2 s is acceptable in a standard setting (the total signature creation time presumably below 3 s). We have to note that the preprocessing phase can be executed in parallel when the user is asked to confirm his will to create a signature – in this case the time overhead is zero, as the user needs a few econds to push the button. The most important message from the experiment is that there is no memory bottleneck on the card to execute the scheme.

Table 1. Timing details for particular steps of the preprocessing phase, the results are given as average over 50 trial executions. For the experiment, each trial involves $\Delta = 100$ (the smart card inserts data in the table A 100 times)

Preprocessing step	Execution time in ms (average over 50 trials)
2	5
3	401
5.1–5.3	1025
5.4	293
Total	1724

6 Related Work

Probably the first attempt to protect the signatories against powerful adversaries capable of deriving signing keys from the public keys are *fail-stop signatures* [11]. In case of a forgery, the legitimate owner of a signing device can break a hard cryptographic problem (that he normally is unable to do) thereby providing an evidence of a forgery. The fail-stop signature schemes are based on the idea that for one public key there is a large number of corresponding secret keys and a cryptanalyst cannot say which of them is kept by the signature creation device of the legitimate signatory. Failure to guess this key and attempt to sign with a different key leads to an evidence mentioned above.

Unfortunately, as far as we know this idea has never been deployed in business practice. One of the reasons might be its target: fail-stop signatures aim to reveal successful cryptanalytic attacks, but do not address the case when the adversary has access to the original signing key (when for instance he retains the keys after generating them for the user). Even if the description of a scheme states that the private key is generated by the user (see e.g. [13]), this is done by the signature creation card, and the card may choose in a way predetermined by the card manufacturer. On the other hand, some of the solutions proposed assume existence of a party that indirectly gets access to the private keys of the users (see e.g. [15]). In our opinion this defers on threat but creates a new one, even more serious. Many of the proposed fail-stop signature schemes are one-time signatures, which makes them useless for standard applications. Last not least, according to the third law of Adi Shamir: "Cryptography is typically bypassed, not penetrated."

A somewhat similar, but simpler idea has been presented in [3]. The idea is to provide exactly two options for creating the ith signature, namely with either $r_i = g^{k_i}$ or $r'_i = g^{k'_i}$. Which option is used depends on the user. In case of device cloning the same random parameter k for computing $r = g^k$ can be used in two different signatures: one created by the legitimate device and one by the cloned device. This leads immediately to revealing the signing key and thereby invalidates all signatures created with this key, including the ones created by the clone. Thereby the cloned devices are of little use. Invalidating all signatures seems to be a drastic measure, however the adversary holding a copy of the private key can always do it by leaking somewhere this key.

An alternative approach based on monitoring creation of electronic signatures has been proposed in [4] as *mediated signatures*. The idea is that in an online environment we need not to rely solely on the signing device: for signature creation it is necessary to involve another party, called *mediator*, that holds a supplementary key for each user. Therefore, the mediator may monitor the activities of the signing device and may inform the user about each signature created by the device. The paper [4] describes a simple implementation based on the RSA signatures, while [10] provides a solution based on Schnorr signatures. This approach has been further extended in [2] by the concept of key evolution. This enables the mediator to detect that it is interacting with two different signing devices attributed to the same user.

Technically, the most related design has been proposed in [8], where the authors aim to detect omissions on the list of electronic signatures corresponding to a single device. This scheme aims to cope with the problem of hiding documents created by a given device leading to cases such as Enron bookkeeping frauds.

Recently, there have been a growing interest on protecting signature schemes against attacks based on malicious software. In [18] a general model for subversion attacks is considered – the one that enables the adversary to replace the original algorithm creating the signature by a malicious one. A general method to leak the key bits is presented (following well-known kleptographic attacks [17] and the algorithm from [8]). An important point is that the change should not detectable by the user not knowing the trapdoor information. The general recommendation of this paper is to use deterministic signature algorithms in order to remove any room for malicious substitution. While this is an convincing argument, it also provides a perfect framework for the adversary holding the signing key – in such a setting no cryptographic procedure can help the owner of the signing key to prove the forgery with stolen original keys.

The paper [19] extends [18] by considering the model where also key generation software may be subverted. The key invention is to design a subversion resilient one-way permutation with a trapdoor. It is based on the idea that first an index r is chosen at random – but instead of using r as an index to one-way permutation, the scheme takes the hash value $r' := h_{\mathrm{SPEC}}(r)$ for computing the index of a permutation in the family of one-way trapdoor permutations. Having a trapdoor permutation it becomes relatively straightforward to construct a subversion resistant signature scheme. However, we again have to stress that while this construction protects against malicious implementations and leaking the signing key via the signatures, it provides no defense against an adversary that gets the private key in a different way – e.g. via retaining r by the service provider. Also, there is a long way from the idea to a real deployment, since widely used standard solutions tend to persist even when facing serious security problems.

Final Remarks

The method presented in this paper is simple enough to be easily incorporated in signing devices as an optional mechanism against frauds concerning electronic signatures. It requires changing neither signature standards nor the software used for signature verification. This is a major practical advantage since otherwise necessity of modifications could prohibit deployment of countermeasures for years.

Deployment of the proposed security measures would substantially improve real reliability and undeniability of digital signatures and should be considered for next generation of cryptographic smart cards. In our opinion, a potential threat of forging signatures by a dishonest technology provider is one of the barriers for widespread use of electronic signatures for legal purposes. In the current situation we feel that it is quite controversial to recommend usage of legally binding electronic signatures.

Even if the probability $\frac{59}{60}$ of detecting a forgery with the original signing keys in a single signature may be viewed as relatively low, this is significantly better than the current state-of-the art. Indeed, today the owner of the signing key is in a legally hopeless situation, if the original keys are used by the forger. Note that for transactions of a high value the security policy may require signing the document several times. If for instance 10 signatures are required, then the probability of a successful attack performed by an adversary holding the original signing key is reduced to $\approx 2^{-59}$.

A nice feature of the method proposed is that it partially defends against weaknesses of (pseudo)random number generators used for signature creation by standard schemes. It does not prohibit an adversary to learn the signing key, however according to Theorem 3 it does not enable the adversary to learn the key V and therefore to create signatures that would pass the forgery detection test.

Of course, the proposed method has also some limitations. If an adversary gets full access to the memory of a signing device $\mathcal{S}ignDev$, including the hidden control key V installed there, then he will be able to mimic the operation of $\mathcal{S}ignDev$ and create signatures that are indistinguishable from the signatures created by $\mathcal{S}ignDev$. A potential solution to this problem would be to use a chain of the hidden control keys similar to the Lamport's chain of hash values. Namely, instead of a single (v, V) one could use (v_i, V_i) for $i = k, k-1, \ldots, 1$, where $v_{i+1} = v_i \cdot \mathrm{Hash}(V_i)$. The signatory would have to keep v_1 only and from time to time to update the hidden public key V by replacing $V = V_j$ by $V = V_{j-1}$.

Finally, we have to warn the user that the proposed solution does not protect against an adversary who has gained control over his PC (or the browser). The problem is that the smart card has no display and the user cannot be sure which document has been signed following his request.

References

1. Becker, G.T., Regazzoni, F., Paar, C., Burleson, W.P.: Stealthy dopant-level hardware Trojans: extended version. J. Cryptogr. Eng. **4**(1), 19–31 (2014)
2. Błaśkiewicz, P., Kubiak, P., Kutyłowski, M.: Digital signatures for e-government - a long-term security architecture. In: Lai, X., Gu, D., Jin, B., Wang, Y., Li, H. (eds.) e-Forensics 2010. LNICSSITE, vol. 56, pp. 256–270. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23602-0_24
3. Błaśkiewicz, P., Kubiak, P., Kutyłowski, M.: Two-head dragon protocol: preventing cloning of signature keys. In: Chen, L., Yung, M. (eds.) INTRUST 2010. LNCS, vol. 6802, pp. 173–188. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25283-9_12
4. Boneh, D., Ding, X., Tsudik, G., Wong, C.M.: Instantenous revocation of security capabilities. In: USENIX Security Symposium (2001)
5. Goyal, V., O'Neill, A., Rao, V.: Correlated-input secure hash functions. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 182–200. Springer, Heidelberg (2011). doi:10.1007/978-3-642-19571-6_12
6. Housley, R.: RFC 2630: cryptographic message syntax. ftp://ftp.rfc-editor.org/in-notes/rfc2630.txt (1999)

7. Kocher, P.C.: Timing attacks on Implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996). doi:10.1007/3-540-68697-5_9
8. Kubiak, P., Kutyłowski, M.: Supervised usage of signature creation devices. In: Lin, D., Xu, S., Yung, M. (eds.) Inscrypt 2013. LNCS, vol. 8567, pp. 132–149. Springer, Cham (2014). doi:10.1007/978-3-319-12087-4_9
9. Nystrom, M., Kaliski, B.: RFC 2985: PKCS #9: selected object classes and attribute types version 2.0. ftp://ftp.rfc-editor.org/in-notes/rfc2985.txt (2000)
10. Nicolosi, A., Krohn, M.N., Dodis, Y., Mazières, D.: Proactive two-party signatures for user authentication. In: NDSS 2003, The Internet Society (2003)
11. Pedersen, T.P., Pfitzmann, B.: Fail-stop signatures. SIAM J. Comput. **26**(2), 291–330 (1997)
12. Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D.A., Weger, B.: Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 55–69. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03356-8_4
13. Susilo, W., Safavi-Naini, R., Gysin, M., Seberry, J.: A new and efficient fail-stop signature scheme. Comput. J. **43**(5), 430–437 (2000)
14. The European Parliament and European Council. Regulation (EU) no 910/2014 of the European Parliament and of the Council on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/EC. Official Journal of the European Union L 257/73 (2014)
15. Yamakawa, T., Kitajima, N., Nishide, T., Hanaoka, G., Okamoto, E.: A short fail-stop signature scheme from factoring. In: Chow, S.S.M., Liu, J.K., Hui, L.C.K., Yiu, S.M. (eds.) ProvSec 2014. LNCS, vol. 8782, pp. 309–316. Springer, Cham (2014). doi:10.1007/978-3-319-12475-9_22
16. Young, A., Yung, M.: Kleptography from standard assumptions and applications. In: Garay, J.A., Prisco, R. (eds.) SCN 2010. LNCS, vol. 6280, pp. 271–290. Springer, Heidelberg (2010). doi:10.1007/978-3-642-15317-4_18
17. Young, A., Yung, M.: Kleptography: using cryptography against cryptography. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 62–74. Springer, Heidelberg (1997). doi:10.1007/3-540-69053-0_6
18. Ateniese, G., Magri, B., Venturi, D.: Subversion-resilient signature schemes. In: Ray, I., Li, N., Kruegel, C. (eds.) CCS 2015, pp. 364–375. ACM, New York (2015)
19. Russell, A., Tang, Q., Yung, M., Zhou, H.-S.: Cliptography: clipping the power of kleptographic attacks. IACR Cryptology ePrint Archive vol. 2015, p. 695 (2015)