Coppercloud: Blind Server-Supported RSA Signatures

Nikita Snetkov[1,2], Jelizaveta Vakarjuk[1,2] and Alisa Pankova[1]
{nikita.snetkov, jelizaveta.vakarjuk, alisa.pankova}@cyber.ee

[1] Cybernetica AS, Mäealuse 2/1, 12618 Tallinn, Estonia
[2] Tallinn University of Technology, Akadeemia tee 15a, 12618, Tallinn, Estonia

Abstract. Privacy-preserving technologies are a well-established area of research for modern digital systems, driven by regulatory mandates, such as the GDPR, and growing demand for the user privacy. A notable example of such technologies is blind signatures: a primitive that supports a creation of privacy-sensitive applications, ranging from secure e-voting and anonymous digital cash to privacy-preserving credentials and identity management systems. In this work, we introduce Coppercloud, a blind server-supported RSA signature scheme designed to enhance privacy in digital identity systems. Coppercloud enables a user to obtain a signature on a message, without revealing its content to the supporting server, while distributing the signing key between the user's device and the supporting server. We formalize the security requirements for blind server-supported signing by defining an ideal functionality $\mathcal{F}^{\mathsf{BlindSpl}}$, and prove that Coppercloud securely realizes this functionality in the Universal Composability (UC) model.

Keywords: Blind Signatures · Universal Composability · RSA · Server-Supported Signatures

1 Introduction

With the adoption of the General Data Protection Regulation (GDPR), which has affected both the European and international digital space, more and more companies and public institutions implement various privacy-preserving technologies (PPTs) in their solutions. In particular, *homomorphic encryption* [43] is deployed to provide private search over online public data [31] and to protect the secrecy of a vote in internet elections [16]; *ring signatures* are used to anonymize transactions in cryptocurrencies [48]; *searchable encryption* [18] could be applied in credential systems [34] and private gun registries [35].

As interest in privacy grows, blind signatures are standing out as a notable example of PPTs. Proposed by [15], blind signatures allow a user to sign a message under a key of some service provider without revealing a message's content to that provider. The widespread usage of blind signatures could be illustrated with the following examples – in i-voting to register an eligible voter preserving their secrecy of the vote [24, 38, 50]; in digital cash and cryptocurrencies to

disburse untraceable e-cash [42, 11, 30, 53]; in a web browser advertisement for the private click measurement [51, 49]; in credential systems to protect private attributes from malicious issuing authorities [47]; for privacy-preserving internet CAPTCHA challenges [20]; for a user authorization in the iCloud Private Relay without a need to provide account information [54].

1.1 Motivation

In April 2024, the European Parliament laid a foundation for the new European Digital Identity Framework [3], with one of its goals to establish development of *digital identity wallets* by EU member states. These wallets enable users to digitally sign and present their credentials, using user-generated digital signatures, to create verifiable proofs of identity. Such wallets could potentially be constructed with the usage of threshold signatures [21] to provide verifiable credential presentation. Among the various techniques for building verifiable credential presentation, threshold signatures [21] offer a robust solution through the distribution of the signing key. Recently, one of the proposals for EU digital wallets [4] was built based on the Smart-ID, a digital signing solution [5]. Smart-ID is an implementation of the Buldas et al. [9] server-supported signature scheme, which could be seen as two-party RSA threshold signing with asymmetric roles: one party acts a lightweight client, the other is a high-performance server. In the proposal, supporting server participating in the signature generation, learns information about the credential that is being presented. This creates a potential privacy leak. A possible solution to this problem is the usage of the blinding to hide a message being signed from the supporting server.

1.2 Contribution

In this work, we address privacy issues of existing RSA-based server-supported signing. Our main contributions in this work are:

1. We formally define a security notion of the *blind server-supported signing* in the universal composability (UC) model [14]. We do so by constructing a new functionality $\mathcal{F}^{\mathsf{BlindSpl}}$ building upon existing ideal functionalities $\mathcal{F}^{\mathsf{gSpl}}$ by Snetkov et al. [46] and $\mathcal{F}^{\mathsf{Pass2Sign}}$ by Camenisch et al. [12].
2. We present a new protocol for blind server-supported RSA signing – *Coppercloud*. It allows a user to obtain a valid RSA signature on a message without revealing its content to the supporting server, while also distributing the signing key between the user's device and the server.
3. We prove that Coppercloud securely realizes the ideal functionality $\mathcal{F}^{\mathsf{BlindSpl}}$ in the UC model.

[3] https://eur-lex.europa.eu/eli/reg/2024/1183/oj/eng
[4] https://cyber.ee/resources/news/estonian-digital-identity-wallet-mvp/
[5] https://www.smart-id.com/

1.3 Related Work

Blind signatures [15, 29, 26, 28, 23] and threshold signatures [44, 19, 22, 36, 40] have been extensively studied over the last two decades. Even several protocols for *threshold blind signatures* have been proposed [37, 17, 33, 39, 3] as well. However, those schemes only allow for the distribution of a signing key between several servers/signing providers. The main downside of such techniques in the context of identity wallets is that they do not give users ownership (even partial) over the signing key.

Server-supported signing usually could be built either by offloading the main computational operations to trusted servers [2, 8, 7, 10], or by using threshold signing [25, 52, 27, 12, 9, 46]. The aforementioned approach by Buldas et al. [9] is used in Smart-ID and could be applied in EUDI wallets. However, their protocol does not hide the content of the signed message from the participating server. An alternative server-supported signing approach is a Pass2Sign [12] protocol which supports blindness against a malicious server. The downside of the Pass2Sign protocol is key generation run locally on the user's device opposed to distributed key generation of Buldas et al [9].

2 Preliminaries

2.1 Notation

We present a necessary notation for this paper in the Table 1.

$y := f(x)$	Assigning output of deterministic function $f(\cdot)$ to a variable y
$y \leftarrow f(x)$	Assigning output of probabilistic/randomized function $f(\cdot)$ to a variable y
$x \leftarrow\!\!\$\ X$	Sampling of an element x uniformly at random from a set X
$\|$	1. Concatenation 2. Parallel execution of protocols
$\Longrightarrow$	Sending set of values from one party to another
$\phi(\cdot)$	Euler's totient function
$\mathbb{Z}_n^\times$	Multiplicative group modulo n
$\mathbb{P}(\lambda)$	Set of primes of size defined by λ

Table 1. General notation

2.2 Universal Composability

The universal composability (UC) framework by Canetti [14] allows one to define and prove the security of protocols in a concurrent execution setting. The core principle behind the UC framework is the ideal-versus-real worlds paradigm. When the protocol is executed in the real world, honest parties run the protocol Π over a network that is controlled by an adversary $\mathcal{A}$. The adversary can also corrupt some parties in the real protocol. To prove that the *real world* protocol Π satisfies certain properties, one would need to define an ideal functionality $\mathcal{F}$. It defines how the system should work in the *ideal world*, where honest parties are modelled as dummy parties, who forward their inputs to this ideal functionality. The ideal-world adversary interacts directly with $\mathcal{F}$, which explicitly specifies what the adversary is allowed to do. The inputs to the protocol are provided by the environment $\mathcal{Z}$ and the corresponding outputs are sent to $\mathcal{Z}$. Then, one is required to construct a proof showing that, for any real adversary, there exists an ideal adversary, such that no polynomial-time environment could distinguish between the communication in the ideal world with the ideal world adversary and the real execution of the protocol with the real adversary.

2.3 Server-Supported Signatures

Server-supported signature as defined by Buldas et al. [9] and Snetkov et al. [46] could be seen as a threshold signing protocol between two parties: client (running on some constrained device like a phone or a tablet) and server, with following properties:

- *Security against offline dictionary attacks*: client's keyshare is generated from a password/PIN and a random bitstring. To guess the password, corrupted client needs to communicate with server with a limited amount of guesses.
- *Clone detection*: server has an ability to detect whether adversary managed to create a copy of client's memory.

2.4 RSA signing

Our protocol produces the so-called *multi-prime RSA signatures* [32]. These are a generalization of standard bi-prime RSA, where instead of a biprime modulus is a product of $v \geq 3$ primes (for Coppercloud, we set $v = 4$). It should be emphasized that multi-prime RSA signatures remain compatible with standard RSA verification algorithms. For the signature padding scheme, we use RSA-PSS [6].

Definition 1 (Multi-prime RSA-PSS). *The RSA Probabilistic Signature Scheme (PSS) defined by $v \geq 3$ prime numbers is called* ***multi-prime RSA-PSS****. It consists of the key generation* RSAKeyGen*, signing* RSASign*, and verification* RSAVerify *algorithms and a supporting padding algorithm* PSSEncode *(Figure 1).*

$\mathsf{RSAKeyGen}(e, v, \lambda)$

1 : $p_1, \ldots, p_v \leftarrow\!\!\$\ \mathbb{P}(\lambda)$
2 : $N := p_1 \cdot p_2 \ldots p_{v-1} \cdot p_v$
3 : if $[\gcd(e, \phi(N)) \neq 1]$, RESTART
4 : $d := e^{-1} \mod \phi(N)$
5 : return $\mathsf{sk} := \langle N, d \rangle$, $\mathsf{pk} := \langle N, e \rangle$

$\mathsf{RSASign}(\mathsf{sk}, \mathsf{msg})$

1 : $\sigma := \mathsf{PSSEncode}(\mathsf{msg})^d \bmod N$
2 : return σ

$\mathsf{RSAVerify}(\mathsf{pk}, \mathsf{msg}, \sigma)$

1 : $\mathsf{salt}_m \| H \| 0xBC \leftarrow (\sigma)^e \bmod N$
2 : $\mathsf{mask} := \mathsf{H}'(H, l_{em} - l_h - 1)$
3 : $\mathsf{salt} := \mathsf{salt}_m \oplus \mathsf{mask}$
4 : $b := H \stackrel{?}{=} \mathsf{H}_2(0^{64} \| \mathsf{H}_1(\mathsf{msg}, l_h) \| \mathsf{salt})$
5 : return b

$\mathsf{PSSEncode}(\mathsf{msg}, l)$

1 : $h := \mathsf{H}_1(\mathsf{msg}, l_h)$
2 : $\mathsf{salt} \leftarrow\!\!\$\ \{0,1\}^{l_s}$
3 : $H := \mathsf{H}_2(0^{64} \| h \| \mathsf{salt})$
4 : $\mathsf{mask} := \mathsf{H}'(H, l_{em} - l_h - 1)$
5 : $\mathsf{salt}_m := \mathsf{salt} \oplus \mathsf{mask}$
6 : return $M := \mathsf{salt}_m \| H \| 0xBC$

Fig. 1. Multi-prime RSA-PSS algorithms

The security of Coppercloud is based on the computational hardness of the one-more-RSA assumption, particularly on the equivalent to it the RSA-ACTI (Alternative Chosen-Target RSA Inversion) problem [4].

Definition 2 (RSA-ACTI [5]). *Let $\mathcal{A}$ be an oracle Turing machine. For the security parameter λ, let the experiment $\mathbf{Exp}^{\mathrm{rsa-acti}}_{\mathcal{A}}(\lambda)$ be defined as follows:*

RSA key pair generation: *The challenger generates an RSA public key (N, e) and secret key d corresponding to the security parameter λ. Define the following oracles:*

1. *The RSA inversion oracle $\mathcal{O}_{\mathcal{I}}(\cdot, N, d)$ that, on input $y \in \mathbb{Z}^*_N$, returns x such that $x^e \leftarrow y \mod N$; i.e., it returns $y^d \mod N$ where $ed \equiv 1 \mod \phi(N)$.*
2. *An oracle $\mathcal{O}_{\mathcal{R}}(\cdot, N)$ that, when queried, issues a random RSA inversion challenge point, i.e. a random element of $\mathbb{Z}^*_N$. Let y_i denote the outcome of the i-th such query.*

$\mathcal{A}$ is invoked: *The challenger invokes $\mathcal{A}^{\mathcal{O}_{\mathcal{I}}(\cdot,N,d), \mathcal{O}_{\mathcal{R}}(\cdot,N)}(N, e)$ and responds to its oracle queries. Eventually, $\mathcal{A}$ terminates.*

$\mathcal{A}$'s success criterion: *Let ℓ be the number of queries $\mathcal{A}$ issued to $\mathcal{O}_{\mathcal{I}}(\cdot, N, d)$. Let $(y_1, \ldots, y_n)$ be the values $\mathcal{A}$ received from $\mathcal{O}_{\mathcal{R}}(\cdot, N)$. Let $(z_1, \ldots, z_n)$ be $\mathcal{A}$'s output. For $1 \leq i \leq n$, z_i is correct if $z_i^e \equiv y_i \mod N$. $\mathcal{A}$ is successful if $|\{i : z_i \text{ is correct}\}| \geq \ell + 1$.*

Let $\mathbf{Adv}^{\mathrm{rsa-acti}}_{\mathcal{A}}(\lambda)$ denote the probability that $\mathcal{A}$ is successful in $\mathbf{Exp}^{\mathrm{rsa-acti}}_{\mathcal{A}}(\lambda)$. The RSA-ACTI problem is hard if for any probabilistic polynomial-time $\mathcal{A}$, $\mathbf{Adv}^{\mathrm{rsa-acti}}_{\mathcal{A}}(\lambda)$ is negligible.

3 Ideal Functionality for Blind Server-Supported Signatures

The ideal functionality for blind server-supported signing $\mathcal{F}^{\mathsf{BlindSpl}}$ is defined in Figures 2–4. The purpose of $\mathcal{F}^{\mathsf{BlindSpl}}$ is to capture the desired behaviour of the distributed privacy-preserving signing system. The ideal functionality is constructed by introducing the blinding properties to the server-supported signing $\mathcal{F}^{\mathsf{gSpl}}$ functionality defined by Snetkov et al. [46]. These changes are inspired by the functionality $\mathcal{F}_{\mathsf{Pass2Sign}}$ introduced by Camenisch et al. [12].

$\mathcal{F}^{\mathsf{BlindSpl}}$ is designed to model an interaction between two distinct parties acting as a *client* and a *server*, while any party could act as a verifier. These two parties have the ability to sign collaboratively upon the client's request, while anyone is able to verify any signature by interacting with $\mathcal{F}^{\mathsf{BlindSpl}}$.

3.1 Corruption Levels and Internal State of $\mathcal{F}^{\mathsf{BlindSpl}}$

In real-world server-supported signing, a client's device is much more prone to various attacks compared to the server. To capture this, $\mathcal{F}^{\mathsf{BlindSpl}}$ has fine-grained corruption levels for the client: $\mathsf{c}_{\mathrm{C}} = 0$ represents *an honest party*; $\mathsf{c}_{\mathrm{C}} = 1$ – *an adversary accessed the client's encrypted memory*; $\mathsf{c}_{\mathrm{C}} = 2$ – *an adversary accessed the client's plaintext memory but has no full control over the client*; $\mathsf{c}_{\mathrm{C}} = 3$ – *an adversary has complete control over the client*. We denote the device's memory between the signing sessions as encrypted memory and the its memory during the signing session as plaintext memory.

We consider an adversary being able to "*escalate*" between different corruption levels which corresponds to real-life scenarios and attacks. The server's corruption levels are more straightforward: $\mathsf{c}_{\mathrm{S}} = 0$ corresponds to an *honest server*, $\mathsf{c}_{\mathrm{S}} = 1$ – full corruption of the server. In some sense, we model a static adversary for the server and an adaptive adversary for the client.

To track different attack scenarios, the internal state of the ideal functionality $\mathcal{F}^{\mathsf{BlindSpl}}$ is initialised with the set of the following flags:

- b_{OK} – governs the signing capability of $\mathcal{F}^{\mathsf{BlindSpl}}$. If $b_{\mathrm{OK}} = 0$, functionality will ignore any signing requests.
- b_{lq} – tracks the origin of the last signing query. $b_{\mathrm{lq}} = 1$ corresponds to the most recent query made by the client, $b_{\mathrm{lq}} = 0$ – by the adversary.
- b_{keyleak} – indicates whether the adversary knows (or is assumed to know) the total private key. If $b_{\mathrm{keyleak}} = 1$, the adversary managed to corrupt both server and client at any positive corruption level, therefore $\mathcal{F}^{\mathsf{BlindSpl}}$ cannot provide any security guarantees.
- b_{KG} – tracks whether a keygen query was made to the ideal functionality.
- ctr_s – tracks the number of successful signing sessions run by both the adversary and the server, storing only this count of valid signatures during verification.

Initialization: On input (init) from $\mathcal{S}$, set $\mathsf{c_S} := \mathsf{c_C} := T := 0$; $b_{\mathrm{lq}} := b_{\mathrm{OK}} := 1$; $\mathsf{ctr}_s := 0$; $\mathsf{Ver} := \perp$. Send (init) to $\mathcal{S}$.
Derived values: $b_{\mathrm{KG}} \equiv [\mathsf{Ver} \neq \perp]$. $b_{\mathrm{keyleak}} \equiv [\mathsf{c_S} = 1] \wedge [\mathsf{c_C} \geq 1]$.
Corrupt server: On input (corrupt-server) from $\mathcal{S}$: set $\mathsf{c_S} := 1$ and send (corrupt-server) to server.
Corrupt client: On input (corrupt-client, ℓ) from $\mathcal{S}$, where $\ell \in \{1, 2, 3\}$:

- If $\mathsf{c_C} > \ell$, ignore this query.
- Set $b_{\mathrm{lq}} := \perp$. If $\mathsf{Ver} = \perp$, set $\mathsf{c_C} := 3$. Otherwise, set $\mathsf{c_C} := \ell$. Send (corrupt-client, ℓ) to client.

Trigger: If b_{keyleak}, then set $\mathsf{c_C} := \max(\mathsf{c_C}, 2)$.
Trigger: If $\mathsf{c_C} \geq 2$ and b_{KG}, then send (corrupt-pin, PIN) to $\mathcal{S}$.
Forwarding: If $\mathsf{c_C} = 3$ [resp. $\mathsf{c_S} = 1$], then inputs from client [resp. server] are forwarded to $\mathcal{S}$; and $\mathcal{S}$ determines the outputs that client [resp. server] receives.

Fig. 2. Initialization and corruption instructions for $\mathcal{F}^{\mathsf{BlindSpl}}$

3.2 Password/PIN Handling

Since the client's key share is generated using some low-entropy password or PIN, $\mathcal{F}^{\mathsf{BlindSpl}}$ has a supporting subroutine process-pin. It verifies a PIN value of length L provided by the client or the adversary. To model protection against password offline-guessing, $\mathcal{F}^{\mathsf{BlindSpl}}$ tracks the number of unsuccessful PIN queries with the counter T. Whenever T becomes greater or equal than the allowed number of attempts T_0, $\mathcal{F}^{\mathsf{BlindSpl}}$ will ignore any signing request and assign $b_{\mathrm{OK}} := 0$.

3.3 Clone detection

To model the server's ability to detect and prevent illegitimate copies of the client's memory, $\mathcal{F}^{\mathsf{BlindSpl}}$ has a supporting subroutine clone-check. This subroutine checks if the previous signing query came from the same source. If not, $\mathcal{F}^{\mathsf{BlindSpl}}$ sets $b_{\mathrm{OK}} = 0$.

3.4 Blind Signing

We define three separate signing interfaces in the $\mathcal{F}^{\mathsf{BlindSpl}}$ functionality for different corruption scenarios.

Signing by client and server is initiated by the inputs of both parties, where the message M and the PIN are the client's inputs. Next, the functionality verifies the source of the previous signing query (clone detection) and the PIN. If both checks are successful, a signature σ is requested from the adversary, without giving M as input (ensuring blindness). The message-signature pair is recorded by the functionality.

Signing by the client and the adversary is similar to the previously described interface. The only difference is that there is no clone detection verification. The

process-pin(PIN'):

- If $\mathsf{PIN} = \mathsf{PIN}'$, set $T := 0$ and return execution to the caller. Otherwise, increment T.
- If $T \geq T_0$ then set $b_{\mathrm{OK}} := 0$.
- Return (sign-fail) to the invoker(s).

clone-check(d):

- If $b_{\mathrm{lq}} = 1 - d$, set $b_{\mathrm{OK}} := 0$ and return (sign-fail) to the invoker(s).
- Otherwise, set $b_{\mathrm{lq}} := d$ and return execution to the invoker.

request-sig(sid, M):

- Send (sign-init, sid) to $\mathcal{S}$.
- Upon receiving (signature, sid, σ) from $\mathcal{S}$, check whether $(M, \sigma, 0)$ is already stored. If true, restart request-sig(sid, M).
- Create record $(M, \sigma, 1)$ and return σ.

Fig. 3. Supporting algorithms for ideal functionality $\mathcal{F}^{\mathsf{BlindSpl}}$

reason is that the malicious server should be responsible for running this check and we cannot expect the adversary to run it correctly.

Finally, there is signing by adversary and the server interface, which depends on the client's level of corruption. For the first corruption level (client's encrypted memory), the functionality verifies the source of the last signing query and the PIN' correctness. If both checks are successful, the corruption level increases. For the second corruption level (the client's plaintext memory but no full control over the client), only the source of the last signing query is verified, as in this case we assume the adversary already knows the correct PIN. For the third corruption level, no additional checks are performed as the adversary has full control over the client. In all three cases, the functionality proceeds by asking the adversary to create σ. For the case where the client is malicious, we do not save signature message pairs to the functionality during the signing. The reason is that the adversary may just ignore the M input and choose something else instead during the signing. We keep a counter of successful signing sessions ctr_s. Next, we save pairs during the verification procedure and for signatures generated by the adversary (the ones that have not yet been saved to the functionality), we let the adversary decide the outcome of the verification. For unforgeability, we need to make sure that we save only the ctr_s valid pairs as this is the number of successful signing sessions.

3.5 Verification

The verification procedure in $\mathcal{F}^{\mathsf{BlindSpl}}$ corresponds to the standard way of performing verification in the UC model [13]. The verification request is performed

Key Generation: On input $(\mathsf{keygen}, \mathsf{sid}, L, T_0, \mathsf{PIN})$ from client and $(\mathsf{keygen}, \mathsf{sid}, T_0)$ from server, if Ver is already recorded, or sid is not unique, or the values T_0 differ for client and server, or $\mathsf{PIN} \notin \{0, \ldots, L-1\}$, ignore this query. Otherwise:

- Send $(\mathsf{keygen\text{-}init}, \mathsf{sid})$ to adversary $\mathcal{S}$.
- If $\mathcal{S}$ returns $(\mathsf{keygen\text{-}stop})$, then send $\perp$ to both client and server.
- Upon receiving $(\mathsf{key}, \mathsf{sid}, \mathsf{Ver})$ from $\mathcal{S}$, where Ver is signature verification algorithm, store $(\mathsf{Ver}, L, \mathsf{PIN}, T_0)$ and send $(\mathsf{key}, \mathsf{sid}, \mathsf{Ver})$ to both client and server.

Signing by client and server: On input $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ from the client and $(\mathsf{sign}, \mathsf{sid})$ from the server, ignore, unless $b_{\mathrm{KG}} \wedge b_{\mathrm{OK}}$ and sid is unique. Otherwise:

- Call $\mathsf{clone\text{-}check}(1)$ and $\mathsf{process\text{-}pin}(\mathsf{PIN}')$. If successful, compute σ using $\mathsf{request\text{-}sig}(\mathsf{sid}, M)$.
- If $\sigma \neq \perp$, output $(\mathsf{sign\text{-}success}, \mathsf{sid}, M, \sigma)$ to client, and $(\mathsf{sign\text{-}success}, \mathsf{sid})$ to server. Otherwise, return $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to client and server.

Signing by client and adversary: On input $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ from the client and $(\mathsf{sign\text{-}server}, \mathsf{sid})$ from $\mathcal{S}$, ignore unless $b_{\mathrm{KG}} \wedge [\mathsf{c_S} = 1]$ and sid is unique. Otherwise:

- Call $\mathsf{process\text{-}pin}(\mathsf{PIN}')$ and then compute σ by querying $\mathsf{request\text{-}sig}(\mathsf{sid}, M)$.
- If $\sigma \neq \perp$, output $(\mathsf{sign\text{-}success}, \mathsf{sid}, M, \sigma)$ to client. Otherwise, return $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to both client and $\mathcal{S}$.

Signing by adversary and server: On input $(\mathsf{sign\text{-}client}, \mathsf{sid}, \mathsf{PIN}')$ from $\mathcal{S}$ and $(\mathsf{sign}, \mathsf{sid})$ from the server: ignore unless $b_{\mathrm{KG}} \wedge b_{\mathrm{OK}} \wedge [\mathsf{c_C} > 0]$ and sid is unique. Otherwise go to step $\mathsf{c_C} \in \{1, 2, 3\}$:

1. Call $\mathsf{clone\text{-}check}(0)$ and $\mathsf{process\text{-}pin}(\mathsf{PIN}')$. If successful, set $\mathsf{c_C} := 2$ and go to step 3.
2. Call $\mathsf{clone\text{-}check}(0)$. If successful, go to step 3.
3. Send $(\mathsf{sign\text{-}init}, \mathsf{sid})$ to $\mathcal{S}$ and upon receiving $(\mathsf{sign\text{-}success}, \mathsf{sid})$, output $(\mathsf{sign\text{-}success}, \mathsf{sid})$ to server. Set $\mathsf{ctr}_s := \mathsf{ctr}_s + 1$. Otherwise, return $(\mathsf{sign\text{-}fail})$ to both $\mathcal{S}$ and server.

Verification: On input $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ from a party P, send $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ to $\mathcal{S}$. Upon receiving $(\mathsf{verified}, \mathsf{sid}, M, \sigma, \phi)$ from $\mathcal{S}$ with $\phi \in \{1, 0\}$:

- if (M, σ, b') is recorded, then set $b := b'$
- if $\phi = 1$, (M, σ, b) has not been recorded before, and $b_{\mathrm{keyleak}} = 1$, set $\mathsf{ctr}_s := \mathsf{ctr}_s - 1$:
  - if $\mathsf{ctr}_s \geq 0$, record $(M, \sigma, b := \phi)$.
  - if $\mathsf{ctr}_s < 0$, set $b := 0$ and record (M, σ, b)
- else, set $b := \phi$ and record (M, σ, b).
- output $(\mathsf{verified}, \mathsf{sid}, M, \sigma, b)$.

Fig. 4. Ideal functionality for blind server-supported signing $\mathcal{F}^{\mathsf{BlindSpl}}$

through the adversary. For the signatures that have not been recorded by the functionality, the adversary decides what the outcome of the verification will be. For the signatures created by the honest client, the functionality decides the verification outcome as it has (M, σ, b) already recorded. For the malicious client case, we let the adversary decide the outcome of the verification as the signature itself was created by the adversary. Additionally, for the signatures created solely by the adversary, we let the adversary decide the verification outcome.

4 Coppercloud: Blind Server-Supported RSA Signing

In this section, we present *Coppercloud*, a two-party protocol between a client and a server, which allows both to collaboratively generate an RSA signing key and create multi-prime RSA signatures under this key. Signing has a blinding mechanism identical to RSA-BSSA [41], which hides the message's content from the supporting server.

4.1 Key Generation

As input, both parties receive fixed RSA public exponent e and security parameter λ. As additional input, the server receives a number of attempts T_0. The client samples two prime numbers (corresponding to the chosen λ-bit security) and computes a modulus such that $n_1 := p_1 \cdot q_1$. Next, they compute a private exponent $d_1 := e^{-1} \mod \phi(n_1)$. Since we assume the user's provided PIN to lack sufficient entropy, the client generates a bitstring $u \leftarrow\!\!\$\ \{0,1\}^\lambda$. Based on the provided PIN and u, the client then calculates two additive shares as $d_1' := \mathsf{genShare}(u, \mathsf{PIN}, n_1)$ and $d_1'' := d_1 - d_1'$. The $\mathsf{genShare}(\cdot)$ algorithm could be seen as a special deterministic function that generates a unique element of $\mathbb{Z}_{n_1}^\times$ for a given input. It could be instantiated with a password-based extendable-output function (for more implementation details see [9, 45]). Finally, the client sends $\langle n_1, d_1'' \rangle$ to the server. We need to ensure that the value d_1' that can be derived through trying out different PINs is not verifiable under pk_1. Since it is only one share of the corresponding private key and the second share is stored on the server, we ensure protection against offline guessing attacks.

$\mathsf{genShare}(u, \mathsf{PIN}, \mathsf{size})$

```
1 : for i ∈ {0, 1, ..., 255} do
2 :     x := F_PIN(u || i)
3 :     if x < size then
4 :         return x
5 : return FAIL
```

Fig. 5. Supporting algorithm $\mathsf{genShare}$

$\mathsf{KeyGen}_C(\lambda, e, \mathsf{PIN})$

1 : $p_1, q_1 \leftarrow\!\!\$\ \mathbb{P}(\lambda)$
2 : $n_1 := p_1 \cdot q_1$
3 : $d_1 := e^{-1} \mod \phi(n_1)$
4 : $u \leftarrow\!\!\$\ \{0,1\}^\lambda$
5 : $d_1' := \mathsf{genShare}(u, \mathsf{PIN}, n_1)$
6 : $d_1'' := (d_1 - d_1') \bmod \phi(n_1)$
7 : $\Longrightarrow S : \langle d_1'', n_1 \rangle$
8 : $S \Longrightarrow : \langle n, w \rangle$
9 : $\mathsf{if}\ [n_1 \nmid n], \perp$
10 : $w' := w$
11 : $\mathsf{store}\ \langle n, n_1, u, w' \rangle$
12 : $\mathsf{return}\ \mathsf{pk} := \langle n, e \rangle$

$\mathsf{KeyGen}_S(\lambda, e, T_0)$

1 : $p_2, q_2 \leftarrow\!\!\$\ \mathbb{P}(\lambda)$
2 : $n_2 := p_2 \cdot q_2$
3 : $d_2 := e^{-1} \mod \phi(n_2)$
4 : $C \Longrightarrow : \langle d_1'', n_1 \rangle$
5 : $n := n_1 \cdot n_2$
6 : $w \leftarrow\!\!\$\ \{0,1\}^\lambda$
7 : $\Longrightarrow C : \langle n, w \rangle$
8 : $T := 0$
9 : $\mathsf{store}\ \langle n, n_1, n_2, d_1'', d_2, w, T, T_0 \rangle$
10 : $\mathsf{return}\ \mathsf{pk} := \langle n, e \rangle$

Fig. 6. Coppercloud Key Generation Protocol

Parallel to the client's execution, the server samples two primes p_2, q_2 and computes a modulus n_2. Upon receiving $\langle n_1, d_1'' \rangle$, the server computes the total modulus $n := n_1 \cdot n_2$ and samples clone detection bitstring w. The clone detection bitstring is introduced to track the source of the last signing query. If the client's memory gets leaked, in the scenario where both the adversary and the client will try to initiate signing with the same string w, it will be detected by the server. Then the server transfers $\langle n, w \rangle$ to the client. Finally, the server initialises a number of failed attempts as $T := 0$ and stores values $n, n_1, n_2, d_1'', d_2, w, T, T_0$.

The client, upon obtaining $\langle n, w \rangle$, verifies whether the total modulus n is divisible by n_1. Upon the failure the key generation is aborted, otherwise the client securely deletes all the values except n, n_1, u, w. The public key is defined as $\mathsf{pk} := (n, e)$. The full key generation protocol's description (with $\mathsf{genShare}(\cdot)$ supporting algorithm) is given in Figures 5–6.

4.2 Signing

The client starts by deriving the key share d_1' using the PIN supplied by the user. Next, the client blinds the encoded message msg to obtain m_*. The blinded message is used to derive the client's partial signature share y. A message, consisting of the client's partial signature y, the blinded message m_*, and the clone-detection bitstring w', is forwarded to the server.

Upon receiving the client's message, the server verifies the clone-detection bitstring and checks whether the client has not reached a threshold T_0 of failed signing attempts. If both checks succeed, the server completes the client's signature share σ_1 using the stored share d_1'' and verifies the received signature share. Successful verification of the client's signature share indicates to the server that

$\mathsf{Blind}(M)$

1 : $r \leftarrow\!\!\$\ \mathbb{Z}_{n_1}^{\times}$ s.t. $\mathsf{gcd}(r, n_1) = 1$
2 : return $(r, M \cdot r^e \bmod n_1)$

$\mathsf{Sign}_C(\mathsf{msg}, \mathsf{PIN}, u, w', n_1, n)$

1 : $d_1' := \mathsf{genShare}(u, \mathsf{PIN}, n_1)$
2 : $r, m_* := \mathsf{Blind}(\mathsf{PSSEncode}(\mathsf{msg}))$
3 : $y := (m_*)^{d_1'} \bmod n_1$
4 : $\Longrightarrow S : \langle y, m_*, w' \rangle$
5 : $S \Longrightarrow : \langle \sigma_*, w \rangle$
6 : $w' := w$
7 : $\sigma := \mathsf{UnBlind}(\sigma_*, r, n)$
8 : return σ

$\mathsf{UnBlind}(\sigma_*, r, n)$

1 : $\sigma := \sigma_* \cdot r^{-1} \bmod n$
2 : return σ

$\mathsf{Sign}_S(d_2, d_1'', n_2)$

1 : $C \Longrightarrow : \langle y, m_*, w' \rangle$
2 : if $\left[w' \neq w \vee T \geq T_0\right], \perp$
3 : $\sigma_1 := {m_*}^{d_1''} \cdot y \bmod n_1$
4 : if $\sigma_1^e \not\equiv m_* \pmod{n_1}$ then
5 : $\quad T \leftarrow T + 1$ and RESTART
6 : $\sigma_* \leftarrow \mathsf{CRT}_{n_1,n_2}(\sigma_1, {m_*}^{d_2} \bmod n_2)$
7 : $w \leftarrow\!\!\$\ \{0,1\}^{\lambda}$
8 : $T := 0$
9 : $\Longrightarrow C : \langle \sigma_*, w \rangle$

Fig. 7. Coppercloud Signing Protocol

the user entered the correct PIN (and the correct d_1' was derived). Then the server proceeds to generate their signature share σ_2 and combine both shares using the Chinese Remainder Theorem (CRT) to obtain σ_*. Next, the server generates an updated clone-detection bitstring w, sending it together with the signature σ_* to the client.

The client unblinds the server's signature and verifies the resulting signature after receiving the response from the server. Should the signature be valid, the client outputs σ. The client saves the new clone-detection bitstring $w' := w$. An detailed description of Coppercloud signing protocol is given in Figure 7.

5 Verification and Efficiency

RSA signatures produced by Coppercloud pass the standard RSA-PSS verification algorithm RSAVerify (Figure 1). The Introduced changes to Buldas et al. [9] protocol do not affect the key generation. The blinding operation incurs negligible overhead to the signing operation and only affects client-side execution [1].

6 Security

6.1 Correctness

Let us prove the correctness of the composition operation for blind signatures.

According to the signing protocol (Figure 7):

$$m_* = M \cdot r^e \mod n_1$$
$$\sigma_1 = m_*^{d_1''} \cdot y \mod n_1 = m_*^{d_1' + d_1''} \mod n_1 = m_*^{d_1} \mod n_1$$
$$\sigma_2 = m_*^{d_2} \mod n_2$$

The Chinese Remainder Theorem (CRT) allows us to compose both signatures shares σ_1, σ_2 under the common modulus $n_1 \cdot n_2$ (given $y_i = n_i^{-1} \mod n_{3-i}$):

$$n = n_1 \cdot n_2$$
$$\sigma_* = \mathsf{CRT}(\sigma_1, \sigma_2) = \sigma_1 \cdot n_2 \cdot y_2 + \sigma_2 \cdot n_1 \cdot y_1 \bmod n$$

Then, we have:

$$\sigma_1 \cdot n_2 \cdot y_2 + \sigma_2 \cdot n_1 \cdot y_1 \mod n =$$
$$M^{d_1} \cdot r^{ed_1} \cdot n_2 \cdot y_2 + M^{d_2} \cdot r^{ed_2} \cdot n_1 \cdot y_1 \bmod n =$$
$$r \cdot (M^{d_1} \cdot n_2 \cdot y_2 + M^{d_2} \cdot n_1 \cdot y_1) \mod n =$$
$$r \cdot \sigma \mod n$$

Therefore, the CRT-based composition used by Coppercloud is correct □

6.2 On denial-of-service attacks

Denial-of-service attacks are generally considered to fall outside the fundamental scope of cryptographic security models. However, to satisfy meticulous reader, we shall discuss potential denial-of-services attacks towards Coppercloud and other server-supported signing protocols.

One attack surface in Coppercloud arises from the exploitation of the clone-detection bitstring w. One possible attack is one in which an adversary observing client-server communication could read the updated w sent by the server, then "*race*" the client by submitting w first, which would cause the server to detect the legitimate client as a "*clone*", causing subsequent signing requests from both the client and the adversary to be rejected.

The key observation – this attack does not allow the adversary to produce any forgeries, which is the primary security goal of $\mathcal{F}^{\mathsf{BlindSpl}}$. To mitigate such attacks, solutions exist as Google PlayProtect [6] that allow the server to attest the client's device.

Another issue occurs if Coppercloud is implemented straightforwardly: clone-detection bitstring, in a sense, makes the protocol stateful for both the client and the server. This could theoretically create synchronization-failures and prevent concurrent signing sessions. This problem is already addressed in Smart-ID through the introduction of retransmission nonces and freshness tokens [7].

[6] https://developers.google.com/android/play-protect
[7] https://github.com/SK-EID/smart-id-documentation/tree/api-v2

6.3 Security of Coppercloud

Since our goal is to prove the security of the blind server-supported signature, we intentionally prevent the server from learning the content of the messages it helps to sign. Therefore, in the security proof we are not expecting adversary to be able to forge a signature on a previously unsigned message, as the server does not know the messages it has already signed. Instead, we want to achieve *one-more unforgeability*, where $\mathcal{A}$ (corrupted client) interacts with the server across n blind-signing sessions. The adversary succeeds in winning if it can produce valid signatures on $n+1$ distinct messages that pass verification. Therefore, we have to rely on one-more RSA assumption instead of RSA assumption. Since RSA-ACTI is equivalent to the one-more RSA assumption, our security proof therefore relies on the RSA-ACTI assumption.

Theorem 1. *If* $\mathsf{genShare}(\cdot)$ *is a pseudo-random function, then the protocol* $\pi_{\mathsf{Coppercloud}}$ *presented in Figures 6–7 is a secure implementation of* $\mathcal{F}^{\mathsf{BlindSpl}}$ *in the random oracle model under the alternative chosen-target RSA inversion (RSA-ACTI) problem.*

To prove Theorem 1 we construct a simulator *Sim* such that for any adversary $\mathcal{A}$, no probabilistic polynomial-time environment can distinguish an execution with $\pi_{\mathsf{Coppercloud}} \| \mathcal{A}$ from an execution of $\mathcal{F}^{\mathsf{BlindSpl}} \| Sim \| \mathcal{A}$ with more than negligible advantage. The simulator state consists of $\langle \mathsf{c_S}, \mathsf{c_C}, n, n_1, n_2, d_1, d_1', d_1'', d_2, u, w \rangle$. Next we describe how the simulator handles and responds to commands from $\mathcal{F}^{\mathsf{BlindSpl}}$. In the proof, we assume static corruptions for the server and adaptive corruptions for the client. We divide the description of the simulator into the corrupted/honest server sections.

The main idea for the proof is to receive a challenge for the RSA-ACTI problem pk^* and, if the server is honest, the simulator uses pk^* as the honest server's public key share $\mathsf{pk}_2 \leftarrow \mathsf{pk}^*$. For the signing sessions initiated through $\mathcal{F}^{\mathsf{BlindSpl}}$, the simulator emulates the server's part of the protocol without knowing the message and using its access to the RSA inversion oracle $\mathcal{O}_{\mathcal{I}}(\cdot)$. When answering random oracle (RO) queries for encoding, the simulator embeds an RSA challenge y_i in the response. Once $\mathcal{A}$ makes a verification query on (M, σ), if the signature is valid and has not been created while communicating with the simulated server, the RSA challenge is embedded to the message signed but $\mathcal{O}_{\mathcal{I}}(\cdot)$ has not been queried on this input. Hence, simulator can extract a solution to the RSA-ACTI challenge from (M, σ).

If the server is corrupted, the simulator uses pk^* as the honest client's public key share $\mathsf{pk}_1 \leftarrow \mathsf{pk}^*$. For the signing sessions, the simulator does not receive the message M as input; it samples a random signature $\sigma \leftarrow\!\!\$\ \mathbb{Z}_n^*$ and later programs the RO to make sure that the encoded message $M \leftarrow \sigma^e \mod n$ is mapped to the actual message msg requested for signing. Programming of the RO occurs when simulator receives a query with msg as an input.

Proof. We define the simulator as follows:

Honest server The simulator gets input $\mathsf{pk}^* = (N, e)$ from the external challenger and simulates the protocol run as described below.

Corruption: On command Corrupt client from $\mathcal{A}$ before the key generation has started, send $(\mathsf{corrupt\text{-}client}, 3)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.

For the key generation, the simulator uses the challenge public key pk^* as the public key share of the server and generates the values associated with the other party's public-key share.

Key generation: On command $(\mathsf{keygen\text{-}init})$ from $\mathcal{F}^{\mathsf{BlindSpl}}$:

- If the client is honest, simulate both the client and the server. Set $\mathsf{pk}_2 := \mathsf{pk}^*$, initialise the values n_1, d_1, n, u, w according to the protocol. Sample $d_1'' \leftarrow\!\!\$\ \mathbb{Z}_{\phi(n_1)}$. The value d_1' does not get explicitly defined (but it is defined through d_1 and d_1'') and the value d_2 is available in the form of oracle access $\mathcal{O}_{\mathcal{I}}(\cdot)$. Set $\mathsf{pk} := (n, e)$ and $\mathsf{Ver} := \mathsf{pk}$ and send $(\mathsf{key}, \mathsf{Ver})$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
  // The real protocol execution is different from the simulated one in the way how d_1'' is generated. In the real protocol, $d_1'' = d_1 - d_1' \pmod{\phi(n_1)}$, where $d_1 = e^{-1} \pmod{n_1}$ and $d_1' = \mathsf{genShare}(u, \mathsf{PIN}, n_1)$. In the simulated protocol, $d_1'' \leftarrow\!\!\$\ \mathbb{Z}_{\phi(n_1)}$. Since genShare algorithm relies on a PRF which is a bijection, the distribution of the value d_1'' in both cases is identical.
- If the client is corrupted, simulate the server's protocol execution:
  - Set $\mathsf{pk}_2 := \mathsf{pk}^*$.
  - Upon receiving $\langle n_1, d_1'' \rangle$ from $\mathcal{A}$, compute $n := n_1 \cdot n_2$, sample clone detection bitstring $w \leftarrow\!\!\$\ \{0,1\}^\lambda$ and send $\langle n, w \rangle$ to $\mathcal{A}$.
  - Set $\mathsf{pk} := (n, e)$ and $\mathsf{Ver} := \mathsf{pk}$ and send $(\mathsf{key}, \mathsf{Ver})$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
  - Since corrupting client during key generation corresponds to full corruption $\mathsf{c}_\mathrm{C} = 3$, receive also $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{BlindSpl}}$.

  The values d_1, d_1', u remain undefined and d_2 is available in the form of oracle access $\mathcal{O}_{\mathcal{I}}(\cdot)$.
  // The real protocol execution is identical to the simulated one, since all the values are generated in the same way as in the real protocol.

Corruption (after key generation):

- On command Leak encrypted memory from $\mathcal{A}$, ignore this query if $b_{\mathrm{KG}} = 0$. Otherwise, send $(\mathsf{corrupt\text{-}client}, 1)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$ and $\langle w, n_1 \rangle$ to the $\mathcal{A}$. The value u has been initialised during the key generation, send it to $\mathcal{A}$.
- On command Leak unencrypted memory from $\mathcal{A}$, ignore this query if $b_{\mathrm{KG}} = 0$. Otherwise, send $(\mathsf{corrupt\text{-}client}, 2)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$ and $\langle w, n_1 \rangle$ to the $\mathcal{A}$. Receive PIN from $\mathcal{F}^{\mathsf{BlindSpl}}$ and compute d_1' from PIN and u using genShare and send $\langle d_1', u \rangle$ to $\mathcal{A}$.
- On command Corrupt client from $\mathcal{A}$, if keys have been generated, proceed in the same way as when responding to Leak unencrypted memory command. Additionally, redefine $d_1'' := d_1 - d_1'$ for d_1' computed from PIN ans u. // This is needed to ensure correctness of the signatures created during the signing queries
  Otherwise, send $(\mathsf{corrupt\text{-}client}, 3)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.

Signing:

On command Signing by client and server, simulate both client and server without knowing msg, since nobody is corrupted. On input (sign-init, sid) from $\mathcal{F}^{\mathsf{BlindSpl}}$:

- Generate random $\mathsf{salt} \leftarrow\!\!\$\ \{0,1\}^{l_s}$ and follow steps of PSSEncode (Figure 8) to produce encoded message M, challenge y_i and randomness r_i.
- Compute blinded message $(m_*, r) \leftarrow \mathsf{Blind}(M)$, client's signature share $\sigma_1 := m_*^{d_1}$ and set $y := \sigma_1 \cdot m_*^{-d_1''} \mod n_1$.
- Simulate sending $\langle y, m_*, w\rangle$ to server.
- Compute $\sigma_1 := y \cdot m_*^{d_1''} \mod n_1$, get server's signature share σ_2 by querying oracle $\mathcal{O}_{\mathcal{I}}(m_*)$. Combine signature shares $\sigma_* := \mathsf{CRT}(\sigma_1, \sigma_2)$.
- Simulate sending $\langle \sigma_*, w', m_*\rangle$ to client.
- Unblind signature $\sigma := \mathsf{UnBlind}(\sigma_*, r)$ and set $z_i := \sigma / r_i$ and save record (y_i, z_i) in $\mathsf{T}_{\mathsf{RSA}}$.
- Send command $(\mathsf{signature}, \mathsf{sid}, \sigma)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$ and save partial record $(\mathsf{sid}, \perp_m, \sigma, M, \mathsf{salt})$.

  // In the real protocol, $y = m_*^{d_1'}$. In the simulated, $y = \sigma_1 \cdot m_*^{-d_1''}$, where $\sigma_1 = m_*^{d_1}$. In the real protocol, $d_1' = \mathsf{genShare}(u, \mathsf{PIN}, n_1)$ and in the simulated protocol $d_1' = d_1 - d_1''$ since $y = m_*^{d_1} \cdot m_*^{-d_1''} = m_*^{d_1 - d_1''}$. Since genShare algorithm relies on a PRF which is a bijection, the distribution of the value d_1' in both cases is identical. Therefore, the distribution of y is identical in both cases.

On command Signing by adversary and server, simulate server's protocol execution:

- With the first message from $\mathcal{A}$, receive (σ_1', w, m_*) and simulate constructing command $(\mathsf{sign}, \mathsf{sid}, \mathsf{PIN}')$ to $\mathcal{F}^{\mathsf{BlindSpl}}$ depending on the corruption level:
  - If encrypted memory is leaked ($\mathsf{c_C} = 1$), simulator has already initialised bitstring's u value. With the message from $\mathcal{A}$, receive $\sigma_1' := m_*^{d_1'} \mod n_1$. Having u, test through all possible $\mathsf{PIN} \in L$ to find PIN' that produces σ_1' from $d_1' := \mathsf{genShare}(u, \mathsf{PIN}, n_1)$. Set $\mathsf{PIN}' := \mathsf{PIN}$ and send command $(\mathsf{sign}, \mathsf{sid}, \mathsf{PIN}')$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
  - If unencrypted memory is leaked $\mathsf{c_C} = 2$, simulator has already initialised value u and knows PIN. Set $\mathsf{PIN}' := \mathsf{PIN}$ and send command $(\mathsf{sign}, \mathsf{sid}, \mathsf{PIN}')$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
  - If client is fully corrupted $\mathsf{c_C} = 3$, simulator already knows PIN. Set $\mathsf{PIN}' = \mathsf{PIN}$ and send command $(\mathsf{sign}, \mathsf{sid}, \mathsf{PIN}')$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
- Upon command (sign-init, sid) from $\mathcal{F}^{\mathsf{BlindSpl}}$, compute $\sigma_1 := \sigma_1' \cdot m_*^{d_1''} \mod n_1$ using stored d_1'' and $\sigma_2 := m_*^{d_2} \mod n_2$ by querying $\mathcal{O}_{\mathcal{I}}(m_*)$.
- Combine σ_1 and σ_2 to the final (blinded) signature σ_* using Chinese Remainder Theorem.
- Verify σ_*, if it is correct, send command (sign-success, sid) to $\mathcal{F}^{\mathsf{BlindSpl}}$.

  // The real protocol execution is identical to the simulated one, since all values are generated in the same way as in the real protocol.

On input to oracle H_2 of the form $v_i = (0^{64} \| h_i \| \mathsf{salt}_i)$, if there is no H_i defined proceed as: // This is done to embed RSA challenge to RO responses

1. Get RSA challenge $y_i \leftarrow \mathcal{O}_{\mathcal{R}}(\cdot)$
2. Find $r_i \in \mathbb{Z}_N$ such that, for $w_i = y_i \cdot r_i^e$, $w_i < 2^{k-1}$ and the binary representation of w_i ends in the byte $0xBC$
3. Set $M := w_i$ and parse $w_i := \mathsf{salt}_{m_i} \| H_i \| 0xBC$
4. Set $DB_i := 0^a \| 0x01 \| \mathsf{salt}_i$ and let $\mathsf{mask} \leftarrow DB_i \oplus \mathsf{salt}_{m_i}$
5. Program RO H' such that $\mathsf{H}'(H_i, l) := \mathsf{mask}$. If the value is already defined, fail.
6. Set $\mathsf{H}_2(v_i, l_h) := H_i$, save (i, v_i, r_i, w_i, H_i) and output H_i

Fig. 8. Answering H_2 queries of the form $v_i = (0^{64} \| h_i \| \mathsf{salt}_i)$

On input to oracle H_1 of the form (msg, l): // To make sure that simulated signatures verify correctly

1. If there is record for $h_i = (\mathsf{msg}, l)$, proceed with h_i. Else, sample $h_i \leftarrow\!\!\$\ \{0,1\}^l$.
2. Find non-completed records of the form $(\mathsf{sid}, \perp_{\mathsf{msg}}, \sigma_i, M_i, \mathsf{salt}_i)$.
3. For all i, send $(\mathsf{verify}, \mathsf{sid}, \mathsf{msg}, \sigma_i)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
4. On message $(\mathsf{verify}, \mathsf{sid}, \mathsf{msg}, \sigma_i)$, respond with $(\mathsf{verified}, \mathsf{sid}, \mathsf{msg}, \sigma_i, \phi := 0)$. Upon receiving $(\mathsf{verified}, \mathsf{sid}, \mathsf{msg}, \sigma_i, b = 1)$ for some i (meaning that the message-signature pair has been recorded by the functionality), parse $M_i = \mathsf{salt}_{m_i} \| H_i \| 0xBC$.
5. If there is no such i (the message has not yet been signed), set $h_i \leftarrow \mathsf{H}_1(\mathsf{msg}, l)$ and output h_i. // Only for honest client and honest server case
6. Else, if h_i has not been defined yet, set $h_i \leftarrow \mathsf{H}_1(\mathsf{msg}, l)$. Program RO by setting $H := \mathsf{H}_2(0^{64} \| h_i \| \mathsf{salt})$. Complete signing record by storing msg and output h_i.

Fig. 9. Answering H_1 queries of the form (msg, l).

The simulator makes call to the RSA-ACTI oracle $\mathcal{O}_{\mathcal{I}}(\cdot)$ for each of the signing sessions initiated by the adversary (malicious client).

Verification: If the client is corrupted, functionality has not recorded any (M, σ) pairs. The correct pairs can be recorded in the functionality during the verification procedure. So far, simulator has embedded RSA challenge y_i in each signature (through the RO). Once $\mathcal{A}$ queries verification on (M, σ), if the signature verifies and has not been created while communicating to the simulated server, the RSA challenge is embedded to the signature but the oracle $\mathcal{O}_{\mathcal{I}}(\cdot)$

has not been queried on this input. Hence, simulator can extract answer to the RSA-ACTI challenge from (M, σ).

- Upon receiving message $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ from $\mathcal{F}^{\mathsf{BlindSpl}}$, verify signature using stored $\mathsf{pk} = (e, n)$ following the instructions from Algorithm 1.
- If the signature is correct, set $\phi := 1$. Otherwise, set $\phi := 0$.
  // At this point, it can be either signature created with the help of simulator or by the adversary alone
- Find i such that $\sigma^e \mod N \mod n_1 \equiv w_i$. Fail if such i does not exist.
- Since $w_i = y_i \cdot r_i^e \mod n_1$, $y_i = (\sigma / r_i)^e$. Set $z_i := \sigma / r_i$.
  // This means that inverse for challenge y_i is found
- Store $(y_i, z_i) \in \mathsf{T}_{\mathsf{RSA}}$.
- Let q_R be number of queries simulator made to the challenge oracle $\mathcal{O}_{\mathcal{R}}()$ and $q_I \leq q_R - 1$ be number of queries simulator made to the inversion oracle $\mathcal{O}_{\mathcal{I}}()$. If number of entries in $\mathsf{T}_{\mathsf{RSA}}$ is q_R, output all $z_i \in \mathsf{T}_{\mathsf{RSA}}$ to the external challenger as solution to RSA-ACTI problem.
  // That is, simulator made less queries to the inversion oracle than to the challenge oracle and managed to find solution to all the received challenges
- Output $(\mathsf{verified}, \mathsf{sid}, M, \sigma, \phi)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.
  // The difference between the real and ideal protocol execution is when the simulated verification returns $b := 0$ while the real verification algorithm returns $b := 1$. The only verification query with correct signature to which the simulator responds with 0 is the one that constitutes finding solution to RSA-ACTI problem. This, in turn, means that the adversary made q_I signing queries through the functionality, but managed to produce $q_I + 1$ valid signatures. The real protocol execution will return 1 on verification of $n+1$-th valid signature and the simulated execution returns 0. If adversary manages to differentiate ideal and the real execution, it is able to solve RSA-ACTI problem.

Corrupted server The simulator gets input $\mathsf{pk}^* = (N, e)$ from the external challenger and simulates the protocol run as described below. For the key generation, simulator uses challenge public key pk^* as the public key share of the client and generating the values of the public key share of the other party.

Corruption: On command Corrupt client from $\mathcal{A}$ before the key generation has started, send $(\mathsf{corrupt\text{-}client}, 3)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$. In this case, the adversary $\mathcal{A}$ knows full private key $\langle d, n \rangle$ and can create valid signatures to the messages of its choice. Thus, our scheme does not provide any security guarantees under this setup.

Key generation: On command $(\mathsf{keygen\text{-}init})$ from $\mathcal{F}^{\mathsf{BlindSpl}}$:

- If client is honest, simulate the client's protocol execution:
  - Set $\mathsf{pk}_1 := \mathsf{pk}^*$ and sample randomly $d_1'' \leftarrow\!\!\$\ \mathbb{Z}_{n_1}$.
  - Send $\langle d_1'', n_1 \rangle$ to $\mathcal{A}$.
  - Upon receiving $\langle n, w \rangle$ from $\mathcal{A}$, check whether $n_1 \mid n$, set $\mathsf{pk} := (n, e)$ and $\mathsf{Ver} := \mathsf{pk}$. Send $(\mathsf{key}, \mathsf{Ver})$ to $\mathcal{F}^{\mathsf{BlindSpl}}$.

  The values d_1', d_2, u, n_2 remain undefined and d_1 is available in the form of oracle access $\mathcal{O}_{\mathcal{I}}(\cdot)$.

// The real protocol execution is different from the simulated one in the way how d_1'' is generated. In the real protocol, $d_1'' = d_1 - d_1' \pmod{\phi(n_1)}$, where $d_1 = e^{-1} \pmod{\phi(n_1)}$ and $d_1' = \mathsf{genShare}(u, \mathsf{PIN}, n_1)$. In the simulated, $d_1'' \leftarrow\!\$\ \mathbb{Z}_{n_1}$. Since genShare algorithm relies on a PRF which is a bijection, the distribution of the value d_1'' in both cases is identical.

– If client is corrupted, then the adversary $\mathcal{A}$ knows full private key $\langle d, n\rangle$ and can create valid signatures to the messages of its choice. Thus, our scheme does not provide any security guarantees under this setup.

Corruption (after key generation):

– On command Leak encrypted memory from $\mathcal{A}$, ignore this query if $b_{\mathrm{KG}} = 0$. Otherwise, send $(\mathsf{corrupt\text{-}client}, 1)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$. Since both parties are now corrupted, our scheme does not provide any security guarantees under this setup.
– On command Leak unencrypted memory from $\mathcal{A}$, ignore this query if $b_{\mathrm{KG}} = 0$. Otherwise, send $(\mathsf{corrupt\text{-}client}, 2)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$. Since both parties are now corrupted, our scheme does not provide any security guarantees under this setup.
– On command Corrupt client from $\mathcal{A}$, send $(\mathsf{corrupt\text{-}client}, 3)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$. If keys have been generated, proceed in the same way as when responding to Leak unencrypted memory command.

Signing: On command Signing by client and adversary, simulate the client's protocol execution. Since the client is honest, we need to simulate protocol without knowing message msg:

– Send $(\mathsf{sign\text{-}server}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{BlindSpl}}$ and get $(\mathsf{sign\text{-}init}, \mathsf{sid})$ as response.
– Generate random $\sigma_1 \leftarrow\!\$\ \mathbb{Z}_{n_1}$ and compute $m_* := \sigma_1^e \mod n_1$
– Set $y := \sigma_1 \cdot m_*^{-d_1''} \mod n_1$ and send $\langle y, m_*, w\rangle$ to $\mathcal{A}$.
– Upon receiving (σ_*, r', m_*) from $\mathcal{A}$, verify σ_*
– Generate random $\sigma \leftarrow\!\$\ \mathbb{Z}_n$ such that $M = \sigma^e \mod n$ ends on $0xBC$
– Parse $M = \mathsf{salt}_m \| H \| 0xBC$
– Sample $\mathsf{mask} \leftarrow\!\$\ \{0,1\}^{l_m}$ and compute $\mathsf{salt} := \mathsf{mask} \oplus \mathsf{salt}_m$. Program RO $\mathsf{mask} := \mathsf{H}'(H, l_m)$.
– Send command $(\mathsf{signature}, \mathsf{sid}, \sigma)$ to $\mathcal{F}^{\mathsf{BlindSpl}}$ and save partial record $(\mathsf{sid}, \perp_m, \sigma, M, \mathsf{salt})$

Now, we need to bind specific signature session i to a message msg. On a query to the random oracle H_1 of the form (msg, l), respond as described on Figure 9.

// The simulated execution is different in the way values y, m_*, σ are generated. In the real protocol client's signature is $y = (m_*)^{d_1'} \mod n_1$, where $d_1' = \mathsf{genShare}(u, \mathsf{PIN}, n_1)$. In the simulated protocol, $y = \sigma_1 \cdot m_*^{-d_1''} \mod n_1$, where $\sigma_1 \leftarrow\!\$\ \mathbb{Z}_{n_1}$. Since message m_* in the real protocol is blinded as $m_* = M \cdot r^e \bmod n_1$, where $r \leftarrow\!\$\ \mathbb{Z}_{n_1}^{\times}$, it is indistinguishable from randomly distributed message $m_* = \sigma_1^e \bmod n_1$ for $\sigma_1 \leftarrow\!\$\ \mathbb{Z}_{n_1}$ in the simulated protocol. Thus, signatures are distributed identically. Additionally, programming RO should

succeed as values H, mask are generated randomly during simulation. We need to program it during the signature generation, since H is outputted as part of the signature and adversary can query H′ on this input.

Verification: Upon receiving message $(\mathsf{verify}, \mathsf{sid}, M = \mathsf{msg}, \sigma)$ from $\mathcal{F}^{\mathsf{BlindSpl}}$:

- Verify signature using stored $\mathsf{pk} = (e, n)$ following the instructions from Algorithm 1.
- If there is uncompleted record with $(\mathsf{sid}, \perp_{\mathsf{msg}}, \sigma, M_i, \mathsf{salt}_i)$, program RO as described on Figure 9. // Uncompleted record means that σ was created during one of the "signing by client and adversary" interactions and the simulator does not know actual message that was signed msg. Hence, at this point we need to bind this signature session to a input message msg through programming RO.
- If there is no record with σ and the signature verifies, it means that this signature has been produced without simulator participating. However, RSA challenge is still embedded in the signature as the verification succeeds:
  - Find i such that $\sigma^e \mod N \mod n_1 \equiv w_i$. Fail if such i does not exist.
  - Since $w_i = y_i \cdot r_i^e \mod n_1$, $y_i = (\sigma / r_i)^e$. Set $z_i := \sigma / r_i$.
  - Save $(y_i, z_i) \in \mathsf{T}_{\mathsf{RSA}}$.
  - Let q_R be the number of queries simulator made to the challenge oracle $\mathcal{O}_{\mathcal{R}}(\cdot)$ and $q_I \leq q_R - 1$ be number of queries simulator made to the inversion oracle $\mathcal{O}_{\mathcal{I}}(\cdot)$. If number of entries in $\mathsf{T}_{\mathsf{RSA}}$ is q_R, output all $z_i \in \mathsf{T}_{\mathsf{RSA}}$ to the external challenger as solution to RSA-ACTI problem, set $\phi := 1$.

// For signatures created through the ideal functionality, output of the verification will be the same as in the real protocol. For signatures that adversary managed to create without making queries to the functionality, the output will be different. The simulated protocol execution returns 0 (as the key is not fully corrupted and the record has not been created though the signing interface) and the real protocol execution returns 1. If adversary can distinguish the two worlds in the latter case, it should have solved RSA problem for the RSA challenge embedded through the random oracle.

□

7 Conclusion

We have introduced a new ideal functionality $\mathcal{F}^{\mathsf{BlindSpl}}$, which captures the security properties of blind server-supported signing. By combining the blindness properties of Chaum-style RSA blind signatures with the efficiency of server-supported signatures, we have constructed the Coppercloud protocol. It enables a user to obtain a valid RSA signature on any message without revealing its content to the supporting server, while keeping the signing key distributed between the user's device and the server. We leave the construction of blind server-supported signature schemes based on other number-theoretic problems to future work.

8 Acknowledgements

We thank Mihkel Jaas Karu for their contributions to this project during the spring of 2025. This research has been supported by the Estonian Research Council, Grant No. PRG1780, and by the European Union under Grant Agreement No. 101087529. The views and opinions expressed are, however, those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Bibliography

[1] Amjad, G., Yeo, K., Yung, M.: RSA blind signatures with public metadata. Cryptology ePrint Archive, Paper 2023/1199 (2023), https://eprint.iacr.org/2023/1199

[2] Asokan, N., Tsudik, G., Waidner, M.: Server-Supported Signatures. In: Bertino, E., Kurth, H., Martella, G., Montolivo, E. (eds.) Computer Security — ESORICS 96. pp. 131–143. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)

[3] Baldimtsi, F., Hanzlik, L., Yadav, A.: Non-interactive blind signatures with threshold issuance. Cryptology ePrint Archive, Paper 2026/400 (2026), https://eprint.iacr.org/2026/400

[4] Bellare, Namprempre, Pointcheval, Semanko: The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. Journal of Cryptology **16**(3), 185–215 (Jun 2003). https://doi.org/10.1007/s00145-002-0120-1, https://doi.org/10.1007/s00145-002-0120-1

[5] Bellare, M., Namprempre, C., Pointcheval, D., Semanko, M.: The power of rsa inversion oracles and the security of chaum's rsa-based blind signature scheme. In: Syverson, P. (ed.) Financial Cryptography. pp. 319–338. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-46088-8_25

[6] Bellare, M., Rogaway, P.: Pss: Provably secure encoding method for digital signatures. IEEE P1363a: Provably secure signatures (1998), https://api.semanticscholar.org/CorpusID:15110839

[7] Bicakci, K., Baykal, N.: Server assisted signatures revisited. In: Okamoto, T. (ed.) Topics in Cryptology – CT-RSA 2004. pp. 143–156. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

[8] Bicakci, K., Baykal, N.: Improved server assisted signatures. Computer Networks **47**(3), 351–366 (2005). https://doi.org/https://doi.org/10.1016/j.comnet.2004.08.008, https://www.sciencedirect.com/science/article/pii/S138912860400235X

[9] Buldas, A., Kalu, A., Laud, P., Oruaas, M.: Server-Supported RSA Signatures for Mobile Devices. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer Security – ESORICS 2017. pp. 315–333. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66402-6_19

[10] Buldas, A., Laanoja, R., Truu, A.: A server-assisted hash-based signature scheme. In: Lipmaa, H., Mitrokotsa, A., Matulevičius, R. (eds.) Secure IT Systems. pp. 3–17. Springer International Publishing, Cham (2017)

[11] Camenisch, J., Hohenberger, S., Lysyanskaya, A.: Compact E-Cash. In: Cramer, R. (ed.) Advances in Cryptology – EUROCRYPT 2005. pp. 302–321. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11426639_18

[12] Camenisch, J., Lehmann, A., Neven, G., Samelin, K.: Virtual Smart Cards: How to Sign with a Password and a Server. In: Zikas, V., Prisco, R.D. (eds.)

Security and Cryptography for Networks - 10th International Conference, SCN 2016, Amalfi, Italy, August 31 - September 2, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9841, pp. 353–371. Springer (2016). https://doi.org/10.1007/978-3-319-44618-9_19

[13] Canetti, R.: Universally composable signature, certification, and authentication. In: Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004. pp. 219–233 (2004). https://doi.org/10.1109/CSFW.2004.1310743

[14] Canetti, R.: Universally Composable Security. J. ACM **67**(5), 28:1–28:94 (2020), https://doi.org/10.1145/3402457

[15] Chaum, D.: Blind Signatures for Untraceable Payments. In: Chaum, D., Rivest, R.L., Sherman, A.T. (eds.) Advances in Cryptology. pp. 199–203. Springer US, Boston, MA (1983). https://doi.org/10.1007/978-1-4757-0602-4_18

[16] Cortier, V., Gaudry, P., Glondu, S.: Belenios: A Simple Private and Verifiable Electronic Voting System. In: Foundations of Security, Protocols, and Equational Reasoning: Essays Dedicated to Catherine A. Meadows, pp. 214–238. Springer (2019). https://doi.org/10.1007/978-3-030-19052-1_14

[17] Crites, E., Komlo, C., Maller, M., Tessaro, S., Zhu, C.: Snowblind: A Threshold Blind Signature in Pairing-Free Groups. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023. pp. 710–742. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-38557-5_23

[18] Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. In: Proceedings of the 13th ACM Conference on Computer and Communications Security. p. 79–88. CCS '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1180405.1180417

[19] Damgård, I., Mikkelsen, G.L., Skeltved, T.: On the Security of Distributed Multiprime RSA. In: Lee, J., Kim, J. (eds.) Information Security and Cryptology - ICISC 2014. pp. 18–33. Springer International Publishing, Cham (2015), https://doi.org/10.1007/978-3-319-15943-0_2

[20] Davidson, A., Goldberg, I., Sullivan, N., Tankersley, G., Valsorda, F.: Privacy Pass: Bypassing Internet Challenges Anonymously. Proceedings on Privacy Enhancing Technologies **2018**(3), 164–180 (Apr 2018). https://doi.org/10.1515/popets-2018-0026

[21] Desmedt, Y., Frankel, Y.: Threshold cryptosystems. In: Brassard, G. (ed.) Advances in Cryptology — CRYPTO' 89 Proceedings. pp. 307–315. Springer New York, New York, NY (1990). https://doi.org/10.1007/0-387-34805-0_28

[22] Doerner, J., Kondi, Y., Lee, E., Shelat, A.: Secure Two-party Threshold ECDSA from ECDSA Assumptions. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 980–997. IEEE (2018), https://doi.org/10.1109/SP.2018.00036

[23] Fuchsbauer, G., Wolf, M.: Concurrently Secure Blind Schnorr Signatures. In: Joye, M., Leander, G. (eds.) Advances in Cryptology – EUROCRYPT

2024. pp. 124–160. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-58723-8_5

[24] Fujioka, A., Okamoto, T., Ohta, K.: A practical secret voting scheme for large scale elections. In: Seberry, J., Zheng, Y. (eds.) Advances in Cryptology — AUSCRYPT '92. pp. 244–251. Springer Berlin Heidelberg, Berlin, Heidelberg (1993). https://doi.org/10.1007/3-540-57220-1_66

[25] Ganesan, R.: Yaksha: augmenting kerberos with public key cryptography. In: Proceedings of the Symposium on Network and Distributed System Security. pp. 132–143 (1995). https://doi.org/10.1109/NDSS.1995.390639

[26] Garg, S., Rao, V., Sahai, A., Schröder, D., Unruh, D.: Round optimal blind signatures. In: Rogaway, P. (ed.) Advances in Cryptology – CRYPTO 2011. pp. 630–648. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_36

[27] Gjøsteen, K., Thuen, Ø.: Password-based signatures. In: Petkova-Nikova, S., Pashalidis, A., Pernul, G. (eds.) Public Key Infrastructures, Services and Applications. pp. 17–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[28] Hanzlik, L., Loss, J., Wagner, B.: Rai-Choo! Evolving Blind Signatures to the Next Level. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology – EUROCRYPT 2023. pp. 753–783. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-30589-4_26

[29] Hazay, C., Katz, J., Koo, C.Y., Lindell, Y.: Concurrently-Secure Blind Signatures Without Random Oracles or Setup Assumptions. In: Vadhan, S.P. (ed.) Theory of Cryptography. pp. 323–341. Springer Berlin Heidelberg, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-70936-7_18

[30] Heilman, E., Baldimtsi, F., Goldberg, S.: Blindly Signed Contracts: Anonymous On-Blockchain and Off-Blockchain Bitcoin Transactions. In: Clark, J., Meiklejohn, S., Ryan, P.Y., Wallach, D., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security. pp. 43–60. Springer Berlin Heidelberg, Berlin, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53357-4_4

[31] Henzinger, A., Dauterman, E., Corrigan-Gibbs, H., Zeldovich, N.: Private Web Search with Tiptoe. In: Proceedings of the 29th Symposium on Operating Systems Principles. p. 396–416. SOSP '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3600006.3613134

[32] Hinek, M.J.: On the security of multi-prime rsa. Journal of Mathematical Cryptology **2**(2), 117–147 (2008). https://doi.org/10.1515/JMC.2008.006

[33] Jarecki, S., Nazarian, P.: Adaptively Secure Threshold Blind BLS Signatures and Threshold Oblivious PRF. Cryptology ePrint Archive, Paper 2025/483 (2025), https://eprint.iacr.org/2025/483

[34] Kalu, A., Kus, B.C., Laud, P., Leung, K.L., Snetkov, N., Vakarjuk, J.: CACS: A Cloud Privacy-Preserving Attribute Management System. In: Proceedings of the 18th International Conference on Availability, Reliability and Security. ARES '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3600160.3605022

[35] Kamara, S., Moataz, T., Park, A., Qin, L.: A Decentralized and Encrypted National Gun Registry. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1520–1537 (2021). https://doi.org/10.1109/SP40001.2021.00072

[36] Komlo, C., Goldberg, I.: FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In: Dunkelman, O., Jacobson, Jr., M.J., O'Flynn, C. (eds.) Selected Areas in Cryptography. pp. 34–65. Springer International Publishing, Cham (2021), https://doi.org/10.1007/978-3-030-81652-0_2

[37] Kuchta, V., Manulis, M.: Rerandomizable Threshold Blind Signatures. In: Yung, M., Zhu, L., Yang, Y. (eds.) Trusted Systems. pp. 70–89. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-27998-5_5

[38] Kumar, M., Katti, C.P., Saxena, P.C.: A Secure Anonymous E-Voting System Using Identity-Based Blind Signature Scheme. In: Shyamasundar, R.K., Singh, V., Vaidya, J. (eds.) Information Systems Security. pp. 29–49. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-72598-7_3

[39] Lehmann, A., Nazarian, P., Özbay, C.: Stronger Security for Threshold Blind Signatures. In: Fehr, S., Fouque, P.A. (eds.) Advances in Cryptology – EUROCRYPT 2025. pp. 335–364. Springer Nature Switzerland, Cham (2025). https://doi.org/10.1007/978-3-031-91124-8_12

[40] Lindell, Y.: Fast Secure Two-Party ECDSA Signing. Journal of Cryptology **34**(4) (Sep 2021), http://dx.doi.org/10.1007/s00145-021-09409-9

[41] Lysyanskaya, A.: Security analysis of rsa-bssa. In: Boldyreva, A., Kolesnikov, V. (eds.) Public-Key Cryptography – PKC 2023. pp. 251–280. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-31368-4_10

[42] Lysyanskaya, A., Ramzan, Z.: Group blind digital signatures: A scalable solution to electronic cash. In: Hirchfeld, R. (ed.) Financial Cryptography. pp. 184–197. Springer Berlin Heidelberg, Berlin, Heidelberg (1998). https://doi.org/10.1007/BFb0055483

[43] Rivest, R.L., Adleman, L., Dertouzos, M.L., et al.: On Data Banks and Privacy Homomorphisms. Foundations of secure computation **4**(11), 169–180 (1978)

[44] Shoup, V.: Practical Threshold Signatures. In: Preneel, B. (ed.) Advances in Cryptology — EUROCRYPT 2000. pp. 207–220. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/{10.1007/3-540-45539-6_15}

[45] SK ID Solutions AS: NQ-SK Certification Practice Statement. Certification practice statement, SK ID Solutions AS (Jul 2019), https://www.sk.ee/upload/files/SK-CPS-NON-QUALIFIED%20SMART-ID-EN-v5_0-20190701.pdf

[46] Snetkov, N., Vakarjuk, J., Laud, P.: Universally Composable Server-Supported Signatures for Smartphones. IACR Communications in Cryptology **3**(1) (2026). https://doi.org/10.62056/a3c3wa0kr

[47] Sonnino, A., Al-Bassam, M., Bano, S., Meiklejohn, S., Danezis, G.: Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to

Distributed Ledgers. In: Proceedings 2019 Network and Distributed System Security Symposium. NDSS 2019, Internet Society (2019). https://doi.org/10.14722/ndss.2019.23272

[48] Sun, S.F., Au, M.H., Liu, J.K., Yuen, T.H.: RingCT 2.0: A Compact Accumulator-Based (Linkable Ring Signature) Protocol for Blockchain Cryptocurrency Monero. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer Security – ESORICS 2017. pp. 456–474. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-66399-9_25

[49] Thomson, M.: An Analysis of Apple's Private Click Measurement (2022), https://mozilla.github.io/ppa-docs/pcm.pdf

[50] Willemson, J.: Analyzing and Improving Eligibility Verifiability of the Proposed Belgian Remote Voting System. In: Rios, R., Posegga, J. (eds.) Security and Trust Management. pp. 126–135. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-47198-8_8

[51] Xiao, Y., Du, J., Zhang, S., Zhang, W., Yang, Q., Zhang, D., Kifer, D.: Click Without Compromise: Online Advertising Measurement via Per User Differential Privacy. In: 2025 IEEE Symposium on Security and Privacy (SP). pp. 2919–2937 (2025). https://doi.org/10.1109/SP61157.2025.00187

[52] Xu, S., Sandhu, R.: Two Efficient and Provably Secure Schemes for Server-Assisted Threshold Signatures. In: Joye, M. (ed.) Topics in Cryptology — CT-RSA 2003. pp. 355–373. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)

[53] Yi, X., Lam, K.Y.: A New Blind ECDSA Scheme for Bitcoin Transaction Anonymity. In: Asia CCS '19: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. p. 613–620. Asia CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3321705.3329816

[54] Zohaib, A., Sheffey, J., Houmansadr, A.: Investigating Traffic Analysis Attacks on Apple iCloud Private Relay. In: Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security. p. 773–784. ASIA CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3579856.3595793