Universally Composable Server-Supported Signatures for Smartphones

Nikita Snetkov^{1,2}[0000-0002-1414-2080], Jelizaveta Vakarjuk^{1,2}[0000-0001-6398-3663], and Peeter Laud¹[0000-0002-9030-8142] {nikita.snetkov,jelizaveta.vakarjuk,peeter.laud}@cyber.ee

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

Abstract. Smart-ID is an application for signing and authentication provided as a service to residents of Belgium, Estonia, Latvia and Lithuania. Its security relies on multi-prime server-supported RSA, passwordauthenticated key shares and clone detection mechanism. Unfortunately, the security properties of the underlying protocol have been specified only in "game-based" manner. There is no corresponding ideal functionality that the actual protocol is shown to securely realize in the universal composability (UC) framework. In this paper, we remedy that shortcoming, presenting the functionality (optionally parameterized with a non-threshold signature scheme) and prove that the existing Smart-ID protocol securely realizes it. Additionally, we present a server-supported protocol for generating ECDSA signatures and show that it also securely realizes the proposed ideal functionality in the Global Random Oracle Model (UC+GROM).

Keywords: Smart-ID \cdot SplitKey \cdot universal composability \cdot ECDSA \cdot RSA \cdot server-supported signatures

1 Introduction

Since the proposal by Whitfield Diffie and Martin Hellman [1], digital signatures have been widely incorporated in computer systems to achieve integrity, authenticity and non-repudiation of the users' data. Several technologies and protocols are utilizing digital signatures, including TLS, SSH, PGP and distributed ledgers. They became so widely implemented that the European Union (eIDAS), United States (ESIGN), and Switzerland (ZertES) equated the legal status of digital signatures with handwritten signatures.

With such widespread application of digital signatures, there is an inherent need to store private keys securely. While there are solutions such as Trusted Execution Environments (TEE), Hardware Secure Modules (HSMs) and smart cards, they do not offer much flexibility or scalability. Meanwhile, software-based solutions are easily and ubiquitously deployed, updated across various devices and platforms, including desktop computers, laptops and smartphones. Still,

software-only solutions raise questions about secure storage and usage the private keys.

Security issues of software-only solutions deployed on partially secure platforms (e.g. smartphones) could be mitigated by applying dedicated servers for remote signing. Nevertheless, corresponding service could reduce the control the individuals have over their private keys. The issues of both security and control can be mitigated by using threshold cryptography [2]. One of the examples of such deployed solution which utilises threshold cryptography is Smart-ID³. It provides an authentication and digital signature creation services for over three million users in Baltic states and Belgium. Technology behind Smart-ID, SplitKey⁴, allows to distribute a private key of a user between their smartphone and a central server. The public keys and signatures created via Smart-ID are indistinguishable from standard RSA ones, which makes Smart-ID functionally interchangeable [3, 4] with RSA and provides interoperability with existing protocols and applications. Additionally, it could be argued that Smart-ID is no less secure solution than smartcards for private key storage [5]. Moreover, Smart-ID is considered to be a Qualified Signature Creation Device (QSCD)⁵, i.e. signatures created with Smart-ID are legally equivalent to the handwritten signatures in the European Union.

1.1 Motivation

The mechanism behind Smart-ID was proposed by Buldas et al. [6]. In their paper, authors presented a protocol between a client (a phone) and a server, allowing to produce multi-prime RSA digital signatures [7]. The protocol considers contrasting capabilities of the client and the server, providing several mechanisms that handle them against various levels of compromising the client. Their solution is effective and practical, but its security analysis could be improved. The authors clearly state the security games covering the main properties of the protocol, including unforgeability by an adversary corrupting the server, and infeasibility of offline attacks by an adversary that has obtained client's (weakly) encrypted memory. However, the more "peripheral" properties have not been addressed in the same manner. Lack of analysis lead to an attack by Sarr [8] against the protection mechanism which catches malicious copies of client's memory (fortunately, that attack has been mitigated in the deployed service⁶). Buldas et al. [6] also do not offer a clear description on how the protocol's guarantees degrade if several attacks or unlikely-but-not-implausible events take place together. In order to clarify such details, a security definition in terms of indistinguishability of the real system from some ideal system is necessary. The security should be proved in the universal composability (UC) framework [9], because such a signature-producing protocol is normally used as a subsystem of some larger system.

³ www.smart-id.com

⁴ https://cyber.ee/products/splitkeyplus

⁵ https://www.smart-id.com/e-service-providers/smart-id-as-a-qscd/

⁶ https://www.tuvit.de/fileadmin/Content/TUV_IT/zertifikate/en/9263BE_s.pdf

Also, it is worth to mention that usage of RSA in Smart-ID has some disadvantages compared to existing elliptic curve schemes:

- 1. Larger size of keys: Practical level of *n*-bit security [10] compels generating RSA keys that are relatively larger than ECDSA or EdDSA keys of similar security level. For example, 128 bit security requires generating 3072 bit long keys for RSA and only 256 bit for ECDSA/EdDSA. At the same time, multi-prime RSA schemes, including Smart-ID, require keys of 6144 bits length to achieve plausible security [11]⁷.
- 2. Hybrid signatures for quantum-safe solutions: To provide proper migration from protocols affected by Shor's algorithm [12] to quantum-secure schemes, it is recommend to support hybrid modes [13–15]. Due to the larger sizes of signatures standardised by National Institute of Standards and Technology (NIST) [16, 17], it makes sense to employ post-quantum signatures together with ECDSA/EdDSA rather than RSA signatures during a transition period. For example, German Federal Office for Information Security (BSI) ceases the issuance of RSA certificates starting from 2029 and will only use ECDSA in hybrid solutions ⁸.

Therefore, there is a need to introduce ECDSA-based Smart-ID/SplitKey to either substitute current RSA-based protocol or bring ECDSA-based protocol as an alternative solution for special use cases.

1.2 Related Work

There are several works on distributed biprime RSA [18–22] and multiprime RSA [23, 24]. Even though distributed biprime RSA protocols allow parties to generate standard RSA signature, they are computationally complex and require to rely on additional assumptions and/or cryptographic primitives. On the contrary, even though distributed multiprime RSA protocols produce larger size signatures and public key, they are much computationally elegant and efficient. Moreover, the resulting signatures are still functionally interchangeable, i.e. could be verified with standard software. Two representative RSA-based server-supported protocols are Camenisch et al. [25] and Buldas et al. [6]. Scheme by Camenisch et al. provides client's privacy from the server and protection against offline-guessing attacks, being proven secure in the universal composability (UC) model. Buldas et al. work improves their approach by introducing distributed key generation and clone-detection but, as mentioned above, does give a proof of their scheme to be secure in UC.

A variety of studies were performed on distributed ECDSA signatures: both in area of two-party [26-33] and *t*-out-of-n [34-42] solutions. Yet, to our best knowledge, there is no ECDSA-based protocol which provides similar features as Camenisch et al. [25] or Buldas et al. [6] works.

⁷ https://github.com/SK-EID/smart-id-documentation/wiki/Smart-ID-service-willstart-to-use-6K-RSA-keys

⁸ https://docbox.etsi.org/Workshop/2023/02_QUANTUMSAFECRYPTOGRAPHY/ TECHNICALTRACK/WORLDTOUR/BSI_KOUSIDIS.pdf

1.3 Our contributions:

- We introduce an ideal functionality \$\mathcal{F}^{gSpl}\$ (Section 4), which describes the properties of server-supported signature generation system with the features matching with Smart-ID/SplitKey. We discuss the details of this system, explaining the choices we have made, or justifying why a certain detail can only be fixed in a specific manner.
- 2. We show that Smart-ID/SplitKey based on multiprime RSA securely realizes \mathcal{F}^{gSpl} functionality in UC-model (Section 5).
- We propose a new server-supported signing protocol based on ECDSA scheme (Section 6) and show it to be secure in UC-model. This protocol could be seen as an server-supported variant of Xue et al. [32] and Kocaman et al. [33] works. To achieve UC security, we define another ideal functionality \$\mathcal{F}_{Sig}^{Spl}\$ parameterized by the signature scheme Sig (in this case, ECDSA). The functionality models the generation of public keys and signatures of Sig, distributed in Smart-ID/SplitKey like manner. We show that our distributed ECDSA signing protocol securely realizes \$\mathcal{F}_{ECDSA}^{Spl}\$ (Section 7).
 We show that there exists a trivial protocol that securely realizes \$\mathcal{F}_{S}^{Spl}\$ in \$\mathcal
- 4. We show that there exists a trivial protocol that securely realizes \mathcal{F}^{gSpl} in \mathcal{F}^{Spl}_{Sig} -hybrid model for any UF-CMA secure signature scheme Sig (Sections 4.3 and A).

2 Preliminaries

2.1 Notation

The relation $a \leftarrow A$ denotes sampling an element a uniformly at random from the set A. The symbol \perp is used to indicate a failure or abort. \mathbb{Z} denotes the set of integers, \mathbb{Z}_q – the set of integers modulo q, \mathbb{G} – a group, \mathbb{F} – a finite field. The symbol \mathcal{F} denotes ideal functionality. The symbol \parallel could mean depending on the context: a) parallel execution of Turing machines or protocols b) concatenation of two inputs.

2.2 Universal Composability

The Universal Composability (UC) framework, introduced by Canetti [9, 43], explores the conditions under which a system composed of interactive Turing machines can securely implement another system. Such a system, which may represent either an actual implementation of a cryptographic protocol Π , or an *ideal functionality* \mathcal{F} capturing the security properties that are desired for the protocol to have, offers two interfaces for the outside world. One interface is meant for the *environment* \mathcal{Z} , meant to model everything that the "legitimate" system contains beside the protocol Π . The second interface is meant for the adversary, modelling the weaknesses (actual for protocols, must-be-tolerated ones for ideal functionalities) of analyzed systems.

Definition 1 (UC-secure). Protocol Π securely implements \mathcal{F} if for any (real) adversary \mathcal{A} , there exists an (ideal) adversary \mathcal{S} , such that no environment \mathcal{Z} can distinguish whether it is interacting with $\Pi || \mathcal{A}$ or with $\mathcal{F} || \mathcal{S}$.

Note that Π and \mathcal{F} must offer the same functionality for \mathcal{Z} . An important property of UC is *composability*: if a system Π securely implements \mathcal{F} , and a different system $\Pi' || \mathcal{F}$ securely implements \mathcal{F}' (we say that " Π' securely implements \mathcal{F}' in \mathcal{F} -hybrid model"), then $\Pi' || \Pi$ also securely implements \mathcal{F}' .

2.3 Building Blocks

Definition 2 (Signature scheme). Sig consists of three algorithms: key generation algorithm KGen that on invocation generates a new keypair (pk, sk), signing algorithm Sign that on inputs $\langle sk, M \rangle$ returns a signature σ on the message M, and the verification algorithm Ver that on inputs $\langle pk, M, \sigma \rangle$ either accepts or rejects given signature.

Definition 3 (Signature correctness). A signature scheme is correct if Ver(pk, M, Sign(sk, M)) always accepts, given that (pk, sk) have been output by KGen().

Definition 4 (UF-CMA security). A signature scheme is universally unforgeable under adaptive chosen-message attacks (UF-CMA) if an adversary with pk and with access to a signing oracle Sign(sk, \cdot) has only a negligible chance of producing a message-signature pair (M, σ), such that Ver(pk, M, σ) accepts, but M was not queried to the signing oracle.

Definition 5 ((ℓ, s) -safe prime). A prime number p is an (ℓ, s) -safe prime, if $p = 2ap'_1 \dots p'_k + 1$, where $p'_i > s$ are prime numbers, and $1 \le a \le \ell$.

Definition 6 (Multi-prime RSA [7, 11]). RSA signature scheme defined by $v \ge 3$ prime numbers is called multi-prime RSA. It consists of the following three algorithms: 1–3.

The underlying scheme in Smart-ID/SplitKey takes as a parameter v = 4 (we shall refer to it as 4RSASplit scheme). As the verification algorithm is independent of the number of primes, this RSA variant is compatible with the usual, wide-spread RSA instances.

Definition 7 (ECDSA [44]). ECDSA is defined over an elliptic curve group \mathbb{G} of order q with generator point G. The group \mathbb{G} is defined by an elliptic curve over the finite field \mathbb{F}_p . Additionally, a hash function H is fixed as $H : \{0,1\}^* \to \mathbb{G}$. ECDSA consists of the following three algorithms 4–6.

Random Oracle Model (ROM) is used to simulate the behavior of hash functions in our protocol [46]. To represent the access of random oracle to several parties properly, we use *Restricted Programmable Observable Global Random Oracle* model introduced by Camenisch [47]. The corresponding functionality \mathcal{F}^{rpoRO} is presented in Figure 2.3.

Algorithm 1: RSA KGen (v, λ)

- 1. Sample randomly picks v distinct prime numbers p_1, \ldots, p_v of size determined by the desired security level λ .
- 2. Calculate $N = p_1 \cdots p_v$ and $\phi(N)$, where $\phi(\cdot)$ is Euler's totient function.
- 3. Pick a (small) integer e such that $gcd(\phi(N), e) = 1$ and compute $d = e^{-1}$ (mod $\phi(N)$),
- 4. Output $\mathsf{sk} = \langle N, d \rangle$, $\mathsf{pk} = \langle N, e \rangle$.

Algorithm 2: RSA Sign(sk, M)

- 1. Calculate $\sigma = H(M)^d \mod N$, where $H : \{0,1\}^* \to \mathbb{Z}_N$ is a suitable padding scheme [45].
- 2. Output the signature σ

Algorithm 3: RSA $Ver(pk, M, \sigma)$

- 1. Check whether $\sigma^e \equiv H(M) \pmod{N}$.
- 2. Output 1, if verification succeeds, otherwise output 0

Algorithm 4: ECDSA KGen(\mathbb{G}, G, q)

- 1. Sample randomly a private key $x \leftarrow \mathbb{Z}_q$.
- 2. Calculate $Q = x \cdot G$.
- 3. Output $\mathsf{sk} = \langle G, q, x \rangle$, $\mathsf{pk} = \langle Q, G, q \rangle$.

Algorithm 5: ECDSA Sign(sk, M)

- 1. Sample randomly an ephemeral key $k \leftarrow \mathbb{Z}_q$ and compute a point $R = k \cdot G$.
- 2. Calculate $r = r_x \mod q$, where r_x is the x-coordinate of the point R. 3. Calculate $s = k^{-1}(H(M) + r \cdot x) \mod q$.
- 4. Output a signature $\sigma = \langle r, s \rangle$.

Algorithm 6: ECDSA Ver(pk, M, σ)

- 1. Calculate $v = s^{-1} \mod q$.
- 2. Calculate $u_1 = H(M) \cdot v \mod q$ and $u_2 = r \cdot v \mod q$.
- 3. Calculate a point $R' = u_1 \cdot G + u_2 \cdot Q$.
- 4. Verify that $r'_x \mod q = r$, where r'_x is a x-coordinate of the point R'.
- 5. Output 1, if verification succeeds, otherwise output 0.

 \mathcal{F}^{rpoRO} : Let HT be the hash table that is used to store queries to the oracle. Let prog be a list of programmed values in the oracle. Both are initially empty. The size of the oracle output is ℓ .

<u>Hash</u>: Upon receiving input $(\mathsf{hash}, \mathsf{sid}, m)$, from a party P:

- If there is a record $(m, h') \in \mathsf{HT}$ for some h', set $h \leftarrow h'$.
- Otherwise, sample random $h' \leftarrow \{0,1\}^{\ell}$ and create a record $(m,h') \in \mathsf{HT}$, set $h \leftarrow h'$.
- If this query is made by the adversary S or if sid does not correspond to a session, then add (sid, m, h) to the list of illegitimate queries Q_{sid} of a session sid.
- Output (hash-response, sid, m, h) to party P.

<u>Observe</u>: Upon receiving input (observe, sid) from the adversary S:

- If \mathcal{Q}_{sid} does not exist, then set $\mathcal{Q}_{sid} = \bot$.
- Output (observed-list, Q_{sid}).

Program: Upon receiving (**program-RO**, m, h) s.t. $h \in \{0, 1\}^{\ell}$ from the adversary S:

- If $\exists h' \in \{0,1\}^{\ell}$ s.t. $(m,h') \in \mathsf{HT}$ and $h \neq h'$, ignore this query.
- Create a record $(m, h) \in \mathsf{HT}$ and set an entry of m in list prog.
- Output (program-confirmed).

IsProgrammed: Upon receiving (is-programmed, sid, m) from a party P:

- If sid does not correspond to current session, ignore this query.

- If $m \in \text{prog}$, set b = 1, else b = 0.
- Output (is-programmed, b)

Fig. 1: \mathcal{F}^{rpoRO} , ideal functionality for Restricted Programmable Observable Global Random Oracle (rpGRO)

Zero-knowledge proof (ZKP) is a protocol defined between two parties: prover and verifier. In this protocol prover aims to convince verifier that they know some value w, such that the relation R(x, w) holds. We could instantiate ZKP as a Σ -protocol which is a three message protocol for a binary relation R (Definition 8). We need NIZKP of knowledge of discrete logarithm for our ECDSA server-supported protocol so the parties could prove the possession of their secret shares. The general approach to construct NIZKP is by using Schnorr identification scheme [48] and applying Fiat-Shamir transform [49] to make it non-interactive. However, Extract(·) algorithm cannot be straightforwardly instantiated for Schnorr-based proofs in the UC. The main reason is the rewinding requirement which cannot be done in UC-model security proof. Lysyanskaya and Rosenbloom [50] have shown how to instantiate NIZKP in the UC model using Fischlin transform [51] to obtain extractable and simulatable NIZKPs. We follow their approach and present functionality \mathcal{F}^{NIZKP} (Figure 2.3). One could debate the usage of Fischlin transform incurs additional costs to the protocol, but as

recently pointed out by Chen and Lindell [52] those costs are unnoticeable in practical applications.

Definition 8. $[\Sigma$ -protocol [50]] A Σ -protocol for a relation R and a protocol template $\tau = (\text{Glob-Setup, Commit, Challenge, Respond, Decision})$ is a tuple of efficient procedures $\Sigma_{R,\tau} = (\text{Setup, Prove, Verify, Sim-Setup, Sim-Prove, Extract})$ defined as:

- Glob-Setup $(1^{\lambda}) \rightarrow ppm$: Upon receiving a security parameter 1^{λ} , invoke Setup (1^{λ}) to obtain the public parameters ppm.
- Commit(ppm, x, w) \rightarrow comm: Prover sends to a verifier message comm.
- − Challenge(ppm, x, comm) → chall: Verifier sends to prover random ℓ -bit string chall.
- Response(ppm, x, w, comm, chall) \rightarrow resp: Prover sends to a verifier a reply message resp.
- Decision(ppm, x, comm, chall, resp) $\rightarrow \{0, 1\}$: Verifier decides whether to accept (output a 1) or reject (output a 0) based on the input (ppm, x, comm, chall, resp).
- $\mathsf{Setup}(1^{\lambda}) \to \mathsf{ppm}$: Upon receiving a security parameter, generate a set public parameters ppm which includes the challenge length ℓ .
- Prove((ppm, x, w), (ppm, x)) $\rightarrow \pi$: Upon receiving inputs from a prover (ppm, x, w) and a verifier (ppm, x), run Commit, Challenge, Respond and output $\pi = (\text{comm, chall, resp})$.
- Verify(ppm, x, π) \rightarrow {0, 1}: Upon receiving a proof π for a statement x, run Decision to output {0, 1}.
- Sim-Setup $(1^{\lambda}) \rightarrow (ppm, z)$: Generate parameters ppm and simulation trapdoor z.
- Sim-Prove(ppm, z, x, chall) $\rightarrow \pi$: Upon receiving public parameters ppm, trapdoor z, statement x, challenge *chall*, produce a proof $\pi = (comm, chall, resp)$.
- Extract(ppm, x, π, π') $\rightarrow w$: Given two proofs $\pi = (\text{comm, chall, resp})$ and $\pi' = (\text{comm, chall', resp'})$ for a statement x such that $\text{Decision}(x, \pi) = \text{Decision}(x, \pi') = 1$ and $\text{chall} \neq \text{chall'}$, output a witness w.

For simplicity and convenience, we shall omit ppm parameter as a part of the input.

Multiplicative-to-Additive (MtA) functionality is a protocol/functionality defined between two parties which allows to transform their multiplicative shares into corresponding additive shares. \mathcal{F}^{MtA} functionality (Figure 3) is a main building block to achieve secure redistribution of an (EC)DSA ephemeral key k between server and client. Doerner et al. [29] have shown a protocol that securely realises \mathcal{F}^{MtA} in UC using oblivious transfer [53, 54]. Theoretically, this functionality could be realised in UC using Paillier encryption [55], Castagnos-Laguillaumie encryption [56] and Joye-Libert encryption [57]. We leave secure realization of \mathcal{F}^{MtA} in UC for other primitives out of the scope of this paper.

Universally Composable Server-Supported Signatures for Smartphones

 $\mathcal{F}^{\mathsf{NIZKP}}$: Setup: Upon receiving (setup, sid) from a party P: - If sid does not correspond to a session, ignore this query. - Send (setup, sid) to adversary \mathcal{S} . receiving – Upon from adversary \mathcal{S} the tuple (algorithms, sid, Prove, Verify, Sim-Setup, Sim-Prove, Extract) with algorithm descriptions, store this tuple. <u>**Prove**</u>: Upon receiving (prove, sid, x, w) from a party P: - If $R(x, w) \neq 1$ or sid does not correspond to a session, ignore this query. - Compute π using Sim-Setup and Sim-Prove algorithms. Run Verify (x, π) . If it returns 1, record and output the message (proof, sid, x, π) Otherwise, output (fail). Verify: Upon receiving (verify, sid, x, π) from a party P: - If sid does not correspond to a session, ignore this query. - If $\operatorname{Verify}(x, \pi) = 0$, output (verification, sid, $x, \pi, 0$). - Otherwise if $(proof, sid, x, \pi)$ is already stored, output $(verification, sid, x, \pi, 1)$. - Otherwise, run Extract to compute w. If R(x, w) = 1, output (verification, sid, $x, \pi, 1$). If R(x, w) = 0, output (fail).

Fig. 2: $\mathcal{F}^{\mathsf{NIZKP}}$, ideal functionality for non-interactive zero-knowledge proof of knowledge.

$\mathcal{F}^{\mathsf{MtA}}$:

<u>Init</u>: On an input (init) from parties P_i and P_j :

- Send (request-init) to \mathcal{S} .
- If respond from S is (init-success), store (init-complete) and output it to P_i and P_j . Otherwise, return (init-fail) to P_i and P_j .

Multiply: Upon receiving (input, sid, $a \in \mathbb{Z}_q$) from P_i and (input, sid, $b \in \mathbb{Z}_q$) from P_j :

- Verify existence of $\mathsf{init}\mathsf{-complete}$ in a memory and uniqueness of $\mathsf{sid}.$ If not, ignore this query.
- Send (multiply-share, sid) to \mathcal{S} .
- If respond from S is (multiply-fail), send \perp to P_i and to P_j . Otherwise, proceed.
- Sample random $\alpha \leftarrow \mathbb{Z}_q$
- Calculate $\beta = a \cdot b \alpha \mod q$
- Send (output, sid, $\alpha)$ to P_i and (output, sid, $\beta)$ to P_j

Fig. 3: \mathcal{F}^{MtA} , ideal functionality for transforming multiplicative shares into additive.

3 Server-Supported RSA Signatures

In this section, we describe a server-supported RSA signing protocol proposed by Buldas et al. [6] with improved clone detection algorithm to mitigate Sarr's

9

attack [8]. We define supporting genShare function (Alg. 7), distributed key generation and signing protocols in Figures 4-5. We discuss security of this protocol in Section 5.

The server-supported RSA protocol is run between two parties: client/phone C and server S in the presence of adversary \mathcal{A} . Buldas et al. [6] indicate that since phone's side of the protocol is executed on the smartphone, the cryptographic material on the phone's side may not be protected as strongly as on the server's side. Considering adversary may infect user's phone with malware, there exists a possibility of phone's memory leakage. If it is done while the key generation or signing process are running, adversary could learn unencrypted memory that corresponds to the phone's private key share. If the attack is executed when no aforementioned protocols are running, adversary gets encrypted memory, that corresponds to learning seed u. Clone detection mechanism aims to prevent impersonation attacks when the phone's memory gets leaked to the adversary. In case of abort message \perp sent by the server during signing, the counter is increased T = T + 1. After one successful signature generation, server resets the counter to be T = 0. In any case, if $T = T_0$, server will not participate in any future communication with the phone.

3.1 Key Generation

For key generation, phone C [resp. server S] generates an RSA modulus n_1 [resp. n_2] (of two primes), such that $gcd(\phi(n_1), e) = gcd(\phi(n_2), e) = 1$ for a previously fixed public exponent e. They compute their private exponents as $d_i = e^{-1} \pmod{\phi(n_i)}$. C also generates a random string u and uses it to compute a value $d'_1 \in \mathbb{Z}_{n_1}$ from PIN. S generates a bitstring w for clone detection purposes. Then phone C sends n_1 and $d''_1 \leftarrow (d_1 - d'_1) \mod \phi(n_1)$ to server S; the latter sends back w and n. C verifies whether $n_1 \mid n$. The public key is $\mathsf{pk} = (n_1 \cdot n_2, e)$. C stores n_1, u, w, pk , deleting d_1, d'_1, d''_1 and the prime factors of n_1 . S stores n_1, n_2, d''_1, d_2, w and initializes the wrong PIN counter $T \leftarrow 0$ with the errors' threshold T_0 .

The value d'_1 is generated using the genShare algorithm (Alg. 7) [6]. It expands PIN into a longer value, using the randomness u that will be leaked if \mathcal{A} obtains phone's encrypted memory. The algorithm calls a pseudorandom function Φ that could be instantiated from a block cipher.

3.2 Signing

In order to **sign** a message M using a given PIN', phone C computes $\hat{d}_1 \leftarrow$ genShare $(u, \text{PIN'}, n_1), \sigma'_1 \leftarrow H(M)^{\hat{d}_1} \mod n_1$, and sends (σ_1, w) to S. The latter accepts such signing requests either from C or \mathcal{A} . Server S checks that received w is equal to stored bitstring w, halting if it is not the case. S generates a bitstring w' and sets it to be the new clone detection string w = w'. S computes $\sigma_1 \leftarrow \sigma'_1 \cdot H(M)^{d''_1} \mod n_1, \sigma_2 \leftarrow H(M)^{d_2} \mod n_2$. Then it combines σ_1 and σ_2 to the signature σ using the Chinese Remainder Theorem, and checks that σ is a valid signature for M (using Algorithm 3). If σ is not a valid signature, then S increments T and stops if it becomes larger than T_0 . Otherwise, S sets T := 0, and sends (M, σ) back to the phone C (or adversary \mathcal{A}).

Algorithm 7: genShare(u, PIN, size)

```
Inputs: The maximum size of the output size, password PIN and access to
pseudo-random function (PRF) \Phi:
for j \in \{0, 1, 2, ..., 255\} do
\begin{vmatrix} x \leftarrow \Phi_{\mathsf{PIN}}(u \mid\mid j); & // x \text{ has } \lceil \log_2 \text{ size} \rceil bits
if x < \text{size then}
\mid \text{ return } x;
end
end
return \bot;
```

4 Ideal Functionalities for Server-Supported Signing

In this section, we define two ideal functionalities for server-supported signing: \mathcal{F}^{gSpl} and \mathcal{F}^{Spl}_{Sig} . The first functionality \mathcal{F}^{gSpl} is generic functionality for server-supported signing, which is instantiated for 2 parties for key generation/signing and n parties for verification. The second functionality \mathcal{F}^{Spl}_{Sig} takes as a parameter any UF-CMA secure signature scheme Sig. The functionality \mathcal{F}^{Spl}_{Sig} could be instantiated only for 2 parties. We show relation between both functionalities in Section 4.3

4.1 Ideal Functionality \mathcal{F}^{gSpl}

 \mathcal{F}^{gSpl} is an ideal functionality for server-supported signatures with personal identification number (PIN) handling and clone detection (Fig. 6). It is built upon the ideal threshold signature functionality of Canetti et al. [39], running with the ideal adversary \mathcal{S} . Towards an environment \mathcal{Z} , it offers a functionality supporting an arbitrary number of parties, where two of them have been identified as *the phone* and *the server*, capable of cooperatively producing signatures, while all parties can verify the signatures.

The internal state of $\mathcal{F}^{g\mathsf{S}p\mathsf{I}}$ contains a set of tuples (M, σ, b) , recording all the fixings that $\mathcal{F}^{g\mathsf{S}p\mathsf{I}}$ has done with regards of some σ being (b = 1) or not being (b = 0) a signature of M. This part of $\mathcal{F}^{g\mathsf{S}p\mathsf{I}}$ is identical to ideal functionalities modelling signatures.

Corruption level: The functionality has a non-trivial internal state, keeping track of the corruption levels of parties, and the source of the last signing query (which is necessary for modelling clone detection). The corruption level c_S of the

$\mathbf{KeyGen}(\lambda) :$

- Initialisation:
- Based on security level λ suitable values of ℓ and s are selected.
- Threshold of attempts T_0 and RSA public exponent e values are defined as well.
- The length of $H(\cdot)$ output is defined to be 2λ .

Phone's 1st message:

- a. Phone C randomly samples two (ℓ, s) -safe primes p_1, q_1 .
- b. Phone C calculates modulus $n_1 = p_1 \cdot q_1$. If $gcd(\phi(n_1), e) \neq 1$, C goes to step a.
- c. Phone C calculates private exponent $d_1 = e^{-1} \mod \phi(n_1)$.
- d. C generates random bitstring u and get PIN from phone's user. Then, C computes phone's share $d'_1 = \text{genShare}(u, \text{PIN}, n_1)$.
- e. Phone C calculates server's share $d''_1 = d_1 d'_1 \mod \phi(n_1)$ and sends $\langle d''_1, n_1 \rangle$ to server S.

Server's 1st message:

- a. Upon receiving $\langle d_1'', n_1 \rangle$, server S randomly samples two (ℓ, s) -safe primes p_2, q_2 .
- b. Server S calculates modulus $n_2 = p_2 \cdot q_2$. If $gcd(\phi(n_2), e) \neq 1$, S goes to step a.
- c. Server S calculates total modulus $n = n_1 \cdot n_2$ and private exponent $d_2 = e^{-1} \mod \phi(n_1)$.
- d. S generates a random clone detection bitstring w and sends $\langle n, w \rangle$ to phone C.

Output:

- Phone C verifies that $n_1 \mid n$. If failed, restart the protocol.
- Phone C stores $n_1, u, w, pk = \langle n, e \rangle$ and securely deletes all other values. Server S stores $n_1, n_2, d''_1, d_2, w, T_0, pk = \langle n, e \rangle$.

Corruptions during KeyGen:

- If adversary \mathcal{A} corrupts C, it possesses values, $d_1, d'_1, d''_1, \mathsf{PIN}, u, w, n_1, n_2$.
- If adversary \mathcal{A} corrupts S, it possesses values, d_2, d''_2, w, n_1, n_2 .



server can be either "0" (uncorrupted) or "1" (corrupted) – in the latter case we think of the adversary as being in full control of the server. The corruption level c_P of the phone is more fine-grained: here "0" stands for an uncorrupted phone and "3" means that the adversary has full control over it. There are also intermediate levels: $c_P = 1$ means that the adversary has managed to learn phone's memory sometime between signing sessions, and $c_P = 2$ means that phone's memory has leaked during a signing session. In the former case, the memory has been "encrypted" with a (presumably low-entropy) PIN. In the latter case, the adversary has learned the key material that the phone uses for signing. In neither "1" or "2" cases, the adversary has full control over the phone that would allow adversary to control the messages the phone sends out.

Sign(M, sid): Phone's 1st message:

a. Phone C computes partial private exponent $d'_1 = \text{genShare}(u, \text{PIN}', n_1)$ and partial signature $\sigma'_1 = H(M)^{\hat{d}_1} \mod n_1$.

b. C sends to $\langle \sigma'_1, w \rangle$ to S.

Server's 1st message:

- a. If $T = T_0$, then server S ignores any further communication with C.
- b. Upon receiving $\langle \sigma'_1, w \rangle$, S checks if w sent in the current session by C coincides with their stored copy of w. If not, it halts any further communication with C.
- c. S calculates signatures $\sigma_1 = \sigma'_1 \cdot H(M)_1^{d''} \mod n_1$ and $\sigma_2 = H(M)^{d_2} \mod n_2$
- d. S combines signatures via Chinese Remainder theorem as $\sigma = CRT(\sigma_1, \sigma_2)$ and verifies resulting signature σ . If $Ver(\sigma, M, pk)$, S sends \perp and increase T = T + 1.
- e. Server samples new cloning detection bitstring w', updates it w = w' and sends $\langle \sigma, w' \rangle$ to phone C.

Output:

- Phone C updates cloning detection bitstring w = w' and verifies σ .
- Both parties output a signature with corresponding message $\langle \sigma, M \rangle$.

Corruptions during Sign:

- If adversary \mathcal{A} corrupts S, it possesses values d_2, d_2'', w, n_1, n_2 .
- If adversary A get access to client's encrypted memory, it possesses values u, w, n₁.
 If adversary A get access to client's unencrypted memory, it possesses values u, d'₁,
- w, n_1 .

Fig. 5: Server-supported RSA signing protocol

Tracking origin of queries: The internal state of functionality contains the bit b_{lq} for indicating whether the phone or the adversary made the last signing query. Whenever the phone is corrupted, this bit is set to \bot , allowing the next signing query to come from either of them. Generally, corrupting the phone means reading its memory, and this presumably includes the current value of the bitstring used for clone detection.

Simulating PIN: The internal state of \mathcal{F}^{gSpl} contains the value of the correct PIN and the counter T for wrong guesses by the adversary. It also contains the range of the possible PINs, L and the allowed amount of guesses T_0 . We have to explicitly model the PIN, because it is available to the environment (as opposed to the bitstring used for clone detection). Guessing the PIN is an issue for phone corruption level $c_P = 1$, therefore it has to be modelled since the PIN may have low entropy. If the adversary manages to guess the PIN, then it can decrypt the memory. Hence, upon successful guess, we set corruption level as $c_P := 2$.

Clone detection and offline-guessing attacks protection: Additionally, the internal state of \mathcal{F}^{gSpl} contains the bits b_{OK} and b_{sk} . The first of them being

Functionality \mathcal{F}^{gSpl} :

On startup, set $\mathsf{Ver} := \bot$; $\mathsf{c}_{\mathrm{S}} := \mathsf{c}_{\mathrm{P}} := T := 0$; $b_{\mathrm{lq}} := b_{\mathrm{OK}} := b_{\mathrm{sk}} := 1$. Key Generation: On input (keygen, L, T_0, PIN) from phone and (keygen, T_0) from server: - If Ver is already recorded or T_0 values differ or PIN $\notin L$, ignore this query. - Send (keygen-init) to adversary \mathcal{S} . - Upon receiving (key, Ver) from adversary \mathcal{S} , where Ver is verification algorithm, store (Ver, L, PIN, T_0) and send (key, Ver) to both phone and server. Corrupt server: On input (corrupt-server) from S: set $c_S := 1$ and send (corrupt-server) to server. Corrupt phone: On input (corrupt-phone, ℓ) from S, where $\ell \in \{1, 2, 3\}$: - If $c_{\rm P} > \ell$, ignore this query. - Set $b_{lq} := \bot$. If $\mathsf{pk} \neq \bot$ or $\ell = 3$ then set $\mathsf{c}_{\mathrm{P}} := \ell$. - Send (corrupt-phone, ℓ) to phone. The following commands are ignored, if key generation has not taken place Signing by phone and server: On input (sign, sid, M, PIN') from the phone and (sign, sid)from the server: - If $b_{OK} = 0$ or sid does not correspond to a session, ignore this query. – Call process-pin(PIN') and clone-check(1). If successful, compute σ using request-sig(sid, M). - If $\sigma \neq \perp$, output (sign-success, sid, M, σ) to phone, and (signature, sid, M, σ) to server. Otherwise, return (sign-fail) to phone and server. Signing by phone and adversary: On input (sign, sid, M, PIN') from the phone and (sign-server, sid) from S: – If $c_{\rm S}=0~{\rm or}$ sid does not correspond to a session, ignore this query. - Call process-pin(PIN') and then compute σ by querying request-sig(sid, M). - If $\sigma \neq \bot$, output (sign-success, sid, M, σ) to phone. Otherwise, return (sign-fail) to both phone and \mathcal{S} . Signing by adversary and server: On input (sign-phone, sid, M, PIN') from S and (sign, sid) from the server: if $c_{\rm P} = 0$ or $b_{\rm OK} = 0$ or sid does not correspond to a session, ignore this query. Otherwise go to step $c_{\rm P} \in \{1, 2, 3\}$: 1. Check that PIN = PIN'. If true, then set T := 0, $c_P := 2$ and go to step 2. Otherwise, call $\mathsf{clone-check}(0)$ and return (sign-fail) to $\mathcal S$ and server. 2. Call clone-check(0). If successful, go to step 3. 3. Compute σ using request-sig(sid, M). If $\sigma \neq \bot$, output (signature, sid, M, σ) to server. Otherwise, return (sign-fail) to both S and server. Verification: On input (verify, sid, M, σ) from a party P: - Define the bit b as follows: • If (M, σ, b') is recorded, then set b = b'. • If (M, σ', b') record does not exist for any σ' and $b_{sk} = 1$, then set b = 0. • Else, set $b = \operatorname{Ver}(M, \sigma)$. - Record (M, σ, b) and output (is-verified, sid, M, σ, b) to party P. Triggers: - If $c_{\rm S} = 1$, $c_{\rm P} \ge 1$, and $\text{Ver} \ne \bot$: Send (corrupt-pin, PIN) to S and set $b_{\rm sk} := 0$. - If $c_P \ge 2$ and $\text{Ver} \ne \bot$: Send (corrupt-pin, PIN) to S.

Fig. 6: Ideal functionality for server-supported signing \mathcal{F}^{gSpl}

Supporting Routines for functionality \mathcal{F}^{gSpl} : process-pin(PIN'):

- If PIN = PIN', set T := 0 and return T to the invoker. Otherwise, increment T.
- If $T \ge T_0$ then set $b_{\text{OK}} := 0$.
- Return (sign-fail) to the two parties (out of phone, server, and \mathcal{S}) that initiated the signing.

clone-check(d):

- If $b_{lq} = 1 d$, set $b_{OK} := 0$ and return (sign-fail) to the two parties that initiated the signing.
- Otherwise, set $b_{lq} := d$ and return it to the invoker.

request-sig(sid, M):

- Send (sign-init, sid, M) to \mathcal{S} .
- Upon receiving (signature, sid, σ , M) from S, check whether $(M, \sigma, 0)$ is already stored. If true, restart request-sig(sid, M).
- Check if $Ver(M, \sigma) = 1$. If true, store $(M, \sigma, 1)$ and return σ ; otherwise output \perp .

Fig. 7: Supporting routines/algorithms for ideal functionality \mathcal{F}^{gSpl}

Machine \mathcal{M}_{PP} :

 $\mathcal{M}_{\mathrm{PP}}$ forwards any message from $\mathcal{F}^{\mathsf{gSpl}}$ unchanged to the phone.

Init: On input (init, L, T_0 , PIN) from the phone:

- Generate and store a random permutation π of the set $\{1, \ldots, L\}$
- Submit (keygen, $L, T_0, \pi(\mathsf{PIN})$) to $\mathcal{F}^{\mathsf{gSpl}}$.

Sign: On input (sign, sid, M, PIN') from the phone: submit the command $(sign, sid, M, \pi(PIN'))$ to \mathcal{F}^{gSpl}

Fig. 8: Description of supporting PIN interception machine \mathcal{M}_{PP}

cleared indicates that clone-checking done by the server has found several copies of phone's memory, or the number of PIN guesses has been exceeded. The second of them being set to 0 indicates that the adversary has corrupted the phone and the server in way that adversary may assumed to know the private key. It could be observed that $c_S = c_P = 1$ is sufficient for learning the private key: in this case the adversary can try out all possible PINs and check its guess against the public key pk and server's share of the private key. Moreover, leaking phone's unencrypted memory tells adversary the PIN: in this case, the leak is assumed to also include encrypted memory, thus all possible PINs can again be tried out.

In order to support the validity of the construction of \mathcal{F}^{gSpl} , we show that it satisfies a number of properties analogous to those presented by Buldas et al. [6], and that its clone detection system works:

Theorem 1. An adversary S and an environment Z running in parallel with \mathcal{F}^{gSpl} , corrupting at most one of phone and server, are not able to create a pair (M, σ) , such that a query (verify, sid, M, σ) by some party would return (is-verified, sid, $M, \sigma, 1$), but M was never submitted to \mathcal{F}^{gSpl} either by the phone or by S.

Proof. Let us assume an adversary S produced a pair (M, σ) , such that a query (verify, sid, M, σ) by some party returns (is-verified, $sid, M, \sigma, 1$). According to the definition \mathcal{F}^{gSpl} , verification command returns b = 1 in two cases:

- 1. If a pair (M, σ) was produced and recorded as a response to signing command by ideal functionality \mathcal{F}^{gSpl} .
- 2. If $b_{sk} = 0$ and adversary recorded a pair (M, σ) such that $Ver(\sigma, M) = 1$.

The first case implies that pair was not produced by adversary S alone. Second case implies that both parties are corrupted by adversary which contradicts given assumption. Therefore, under given conditions adversary S cannot produce such pair (M, σ) .

Theorem 2. An adversary S and an environment Z, running in parallel with $\mathcal{M}_{PP} \| \mathcal{F}^{gSpl}$, not corrupting the server, and corrupting the phone to at most level 1, have at most T_0/L probability (where T_0 and L are fixed during the initialization of \mathcal{F}^{gSpl}) of creating a pair (M, σ) , such that a query (verify, sid, M, σ) by some party would return (is-verified, sid, $M, \sigma, 1$), but M was never submitted to \mathcal{F}^{gSpl} by the phone.

Our second theorem defines security of the mechanisms limiting adversary's guesses of the PIN. Note that these mechanisms involve both the counter T in Fig. 6, as well as the clone detection mechanism that is careful to reset that counter. In order to state that theorem, we have to make sure that the environment cannot communicate the PIN to the adversary through some direct channel between them. Hence we are going to introduce the machine $\mathcal{M}_{\rm PP}$ whose task is to permute the possible PINs that the phone submits to $\mathcal{F}^{\mathsf{gSpl}}$. In this way, the "real" PIN is not known by \mathcal{Z} .

We let \mathcal{M}_{PP} be a machine that has an interface which allows it to be placed between the phone (as part of \mathcal{Z}) and the functionality \mathcal{F}^{gSpl} (or any protocol implementing \mathcal{F}^{gSpl}). The machine \mathcal{M}_{PP} is defined in Figure 8.

Proof. Let us assume an adversary S produced a pair (M, σ) , such that a query (verify, sid, M, σ) by some party returns (verified, sid, M, σ , 1). According to the definition \mathcal{F}^{gSpl} , verification command returns b = 1 in two cases:

1. If a pair (M, σ) was produced and recorded as a response to signing command by ideal functionality \mathcal{F}^{gSpl} . In this case, \mathcal{S} should have guessed PIN. \mathcal{S} can make PIN guessing attempts, when $c_P = 1$ with each command, \mathcal{S} can test only one value of the PIN. When \mathcal{S} submits wrong guess, \mathcal{F}^{gSpl} increments counter T until it reaches T_0 and \mathcal{F}^{gSpl} stops. The mechanisms for reducing T without guessing the PIN are the following:

- Increase c_P , after which PIN is revealed to S and wrong guesses of PIN are no longer tracked. We have presumed that this does not happen.
- Let the phone make a signing query. In this case, the clone detection mechanism prevents any further guesses.
- 2. If $b_{sk} = 0$ and adversary recorded a pair (M, σ) such that $Ver(\sigma, M) = 1$ during verification command.

First case implies that M was submitted to $\mathcal{F}^{\mathsf{gSpl}}$ by either phone or \mathcal{S} , where \mathcal{S} could have submitted M to $\mathcal{F}^{\mathsf{gSpl}}$ with probability at most L/T_0 . Second case implies that both parties are corrupted by adversary. Both of these cases contradict with the given assumptions.

Theorem 3. Suppose that an adversary S and an environment Z are running in parallel with \mathcal{F}^{gSpl} , not corrupting the server and corrupting the phone to at most level 2. Suppose that S and Z cause \mathcal{F}^{gSpl} to be initialized, after which (not necessarily immediately) they issue commands for <u>Signing by phone and server</u>, and for <u>Signing by phone and adversary</u> (in either order, with any number of other commands in between), such that between these two sets of commands there is no command to corrupt the phone to level 3. Then, it is impossible for S and Z to produce a pair (M, σ) , such that a query (verify, sid, M, σ) by some party would return (is-verified, sid, $M, \sigma, 1$), but M was not submitted for signing to \mathcal{F}^{gSpl} before the second of the above command sets was issued.

Before providing a proof for Theorem 3, we also want to clarify on how clone detection mechanism works. Buldas et al. [6] specified for their protocol that the random string chosen for clone detection is generated by the phone. Sarr [8] showed how this could give the adversary many attempts at guessing the PIN, by resetting this string after (L - 1) tries to an old value. They proposed an improvement, changing protocol to make server generate the clone detection string. However, Sarr [8] did not provide adequate security property under any model.

Our theorems state such property; in fact, a part of it appears in Thm. 2, stating that the adversary only gets L tries. We also want to state that clone detection is practical security measure even when phone's unencrypted memory has leaked, which is formalized by Theorem 3.

Proof. Let us assume an adversary S produced a pair (M, σ) , such that a query (verify, sid, M, σ) by some party returns (is-verified, $sid, M, \sigma, 1$). According to the definition \mathcal{F}^{gSpl} , verification command returns b = 1 in two cases:

- 1. If a pair (M, σ) was produced and recorded as a response to signing command by ideal functionality \mathcal{F}^{gSpl} .
- 2. If $b_{sk} = 0$ and adversary recorded a pair (M, σ) such that $Ver(\sigma, M) = 1$ during the verification command.

First case implies that M was submitted to \mathcal{F}^{gSpl} by either phone or \mathcal{S} . This suggests that \mathcal{S} could have submitted M to \mathcal{F}^{gSpl} only if phone has not issued signing commands after \mathcal{S} . It would mean that clone detection mechanism is

activated and S cannot issue signing commands anymore. Second case implies that both parties are corrupted by adversary. Both of these cases contradict with the given assumptions.

4.2 Ideal Functionality \mathcal{F}_{Sig}^{Spl}

Next, we define functionality \mathcal{F}_{Sig}^{Spl} (Figure 9) which is less general and more similar to the ideal functionalities defined for threshold ECDSA protocols in the literature [39, 42]. It is parameterized by a signature scheme Sig, and models the distributed key and signature generations of that scheme. There are lot of similarities between \mathcal{F}_{Sig}^{Spl} and \mathcal{F}_{Sig}^{Spl} , including the corruption levels of the phone and the server, the PIN checking and the clone detection mechanisms. Noteworthy differences are in the generation of keys and signatures, where \mathcal{F}_{Sig}^{Spl} uses the methods of Sig, instead of invoking the adversary \mathcal{S} . We also merge the bit b_{OK} together with b_{sk} into b_{OK} . At the point where \mathcal{S} has corrupted the phone and the server so much that it should learn sk, it does actually learn sk.

the server so much that it should learn sk, it does actually learn sk. The analogues of Theorems 1–3 can be stated for \mathcal{F}_{Sig}^{Spl} . With theorems 4–6, we show that our ideal functionality \mathcal{F}_{Sig}^{Spl} has properties similar to $\mathcal{F}_{gSpl}^{gSpl}$, correctly capturing the security requirements that we set for server-supported signature schemes:

Theorem 4. Let Sig be a signature scheme. If Sig is UF-CMA secure, then an adversary S and an environment Z running in parallel with \mathcal{F}_{Sig}^{Spl} , corrupting at most one of phone and server, have a negligible chance of creating a pair (M, σ) such that $\operatorname{Ver}(\mathsf{pk}, M, \sigma)$ accepts, but M was never submitted to \mathcal{F}_{Sig}^{Spl} either by the phone of by S.

Proof. Assume the opposite. We construct a machine $\mathcal{B}^{Sign(sk,\cdot)}(pk)$ that will break the UF-CMA security of Sig. The machine \mathcal{B} internally executes $\mathcal{F}_{Sig}^{Spl} \| \mathcal{Z} \| \mathcal{S}$ except that

- Upon initialization, \mathcal{B} does not make \mathcal{F}_{Sig}^{Spl} generate a keypair for signing and verification. Instead, it uses its argument pk as the public key. There is no private key sk available to \mathcal{B} .
- Upon signature queries, instead of computing σ via Sign(sk, M), the machine \mathcal{B} submits M to its oracle and receives back σ .

As long as both the phone and the server have not been corrupted, the view of $\mathcal{Z}||\mathcal{S}$ in an actual execution with \mathcal{F}_{Sig}^{Spl} is equal to their view when executing inside $\mathcal{B}^{Sign(sk,\cdot)}(pk)$. Hence there is non-negligible chance that $\mathcal{Z}||\mathcal{S}$ construct a pair (M, σ) , such that $Ver(pk, M, \sigma)$ accepts, but M was never an argument to a sign or sign-phone queries. But in this case, the definition of \mathcal{F}_{Sig}^{Spl} shows that Mwas also not submitted to \mathcal{B} 's signing oracle. Hence (M, σ) is a valid forgery. Functionality \mathcal{F}_{Sig}^{Spl} :

On startup, set $\mathbf{p}\mathbf{k} = \mathbf{s}\mathbf{k} = \bot$, $\mathbf{c}_{\mathrm{S}} = \mathbf{c}_{\mathrm{P}} = T = 0$, $b_{\mathrm{lq}} = b_{\mathrm{OK}} = 1$. All commands are ignored if $b_{\mathrm{OK}} = 0$.

Key Generation: On input (keygen, L, T_0, PIN) from the phone and command (keygen, T_0) from the server:

- if $(\mathsf{pk},\mathsf{sk})$ are already defined, or the values T_0 differ, or $\mathsf{PIN} \notin \{1, \ldots, L\}$, then ignore this query.
- Generate (pk, sk) by querying KGen() and send (keygen-start, pk) to S.
- If S returns (keygen-stop), then set $b_{OK} := 0$ and send \perp to both phone and server. - If S returns (keygen-ok), then store L, PIN, and T_0 , and send pk to both phone and server.

Corrupt server: On input (corrupt-server) from S: set $c_S := 1$. Send (corrupt-server) to server.

Corrupting the phone: On input (corrupt-phone, ℓ) from S, where $\ell \in \{1, 2, 3\}$:

- If $c_{\rm P} > \ell$, ignore this query.
- Set $b_{lq} := \bot$. If $\mathsf{pk} \neq \bot$ or $\ell = 3$ then set $\mathsf{c}_{\mathsf{P}} := \ell$.
- Send (corrupt-phone, ℓ) to phone.

The following commands are ignored, if initialization has not taken place Signing by phone and server: On input (sign, sid, M, PIN') from the phone and (sign, sid) from the server:

- If sid does not correspond to a session, ignore this query.
- Call process-pin(PIN') and clone-check(1). If successful, compute σ by calling Sign(sk, M),
- Send (sign-success, sid, M, σ) to phone and (signature, sid, M, σ) to server.

Signing by phone and adversary: On input (sign, sid, M, PIN') from the phone and (sign-server, sid) from S:

- If $c_{\rm S} = 0$ or sid does not correspond to a session, ignore this query.
- Call process-pin(PIN') and then compute σ by calling Sign(sk, M).
- Send (signature, sid, M, σ) to S and (sign-success, sid, M, σ) to phone.

Signing by adversary and server: On input (sign-phone, sid, M, PIN') from S and (sign, sid) from the server: if $c_P = 0$ then ignore. Otherwise go to step $c_P \in \{1, 2, 3\}$:

- 1. Check that $\mathsf{PIN} = \mathsf{PIN}'$. If this is the case then set T := 0, $c_{\mathsf{P}} := 2$ and go to step 2. Otherwise, call clone-check(0) and return (sign-fail) to S and server.
- 2. Call $\mathsf{clone-check}(0)$. If successful, go to step 3.
- 3. Compute σ by calling Sign(sk, M). Send (sign-success, sid, M, σ) to S and (signature, sid, M, σ) to server.

Subroutines:

process-pin(PIN'):

- If PIN = PIN', set T := 0 and return T to the invoker. Otherwise, increment T.
- If $T \geq T_0$ then set $b_{\text{OK}} := 0$.
- Return (sign-fail) to the two parties (out of phone, server, and \mathcal{S}) that initiated the signing.

clone-check(d):

- If $b_{lq} = 1 d$, set $b_{OK} := 0$ and return (sign-fail) to the two parties that initiated the signing.
- Otherwise, set $b_{lq} := d$ and it return to the invoker.

Triggers:

- If $c_{\rm S} = 1$, $c_{\rm P} \ge 1$, and $\mathsf{Ver} \ne \bot$: Send (corrupt-all, PIN, sk) to \mathcal{S} and put $b_{\rm OK} := 0$. - If $c_{\rm P} \ge 2$, and $\mathsf{Ver} \ne \bot$: Send (corrupt-pin, PIN) to \mathcal{S} .

Theorem 5. Suppose that an adversary S and an environment Z running in parallel with $\mathcal{M}_{PP} \| \mathcal{F}_{Sig}^{Spl}$, not corrupting the server and corrupting the phone to at most level 1, have the probability p of producing a pair (M, σ) such that $\operatorname{Ver}(\mathsf{pk}, M, \sigma)$ accepts, but M was never submitted to \mathcal{F}_{Sig}^{Spl} by the phone. If p is significantly greater than T_0/L (where the values T_0 and L are fixed during the initialization of \mathcal{F}_{Sig}^{Spl}), then Sig is not UF-CMA secure.

Proof. We construct a machine $\mathcal{B}^{Sign(sk,\cdot)}(pk)$ that will break the UF-CMA security of Sig. The machine \mathcal{B} executes $\mathcal{M}_{PP}||\mathcal{F}_{Sig}^{Spl}||\mathcal{Z}||\mathcal{S}$ internally, except:

- Upon initialisation, instead of generating keypair as defined in \mathcal{F}_{Sig}^{Spl} , use key pk that was supplied as input to \mathcal{B} . No sk gets generated by \mathcal{F}_{Sig}^{Spl} . Instead of receiving phone supplied PIN \mathcal{T}_{Sig}^{Spl} receives and stores $\sigma(\text{PIN})$ from \mathcal{M}
- receiving phone-supplied PIN, \mathcal{F}_{Sig}^{Spl} receives and stores $\pi(PIN)$ from \mathcal{M}_{PP} . – Upon receiving signing command, instead of computing signature σ via Sign(sk, M), it makes query to the signing oracle that \mathcal{B} has access to and use the obtained σ as the response to the signing command.

Suppose \mathcal{B} produces a verifiable pair (M, σ) , such that M was never submitted to \mathcal{F}_{Sig}^{Spl} by the phone. In order to produce such pair via submitting M to \mathcal{F}_{Sig}^{Spl} by \mathcal{S} , it must guess the PIN. The guessing is enabled if $c_P = 1$ and at each guess, the adversary can test one possible value of the PIN. At each wrong guess, \mathcal{F}_{Sig}^{Spl} increments the counter T of wrong guesses, and stops when it reaches T_0 . The mechanisms for reducing T without guessing the PIN are the following:

- Increase $c_{\rm P}$, after which PIN is revealed to S and wrong guesses of PIN are no longer tracked. We have presumed that this does not happen.
- Let the phone make a signing query. In this case, the clone detection mechanism prevents any further guesses.

Hence the probability of \mathcal{B} producing a verifiable pair (M, σ) , where M was submitted to \mathcal{F}_{Sig}^{Spl} by the adversary, is at most T_0/L . The probability of M being never submitted to the signing oracle is thus at least $(p - T_0/L)$, which is non-negligible.

Theorem 6. Suppose that an adversary S and an environment Z are running in parallel with \mathcal{F}_{Sig}^{Spl} , not corrupting the server and corrupting the phone to at most level 2. Suppose that S and Z cause \mathcal{F}_{Sig}^{Spl} to be initialized (returning public key pk), after which (not necessarily immediately) they issue commands for Signing by phone and server, and for Signing by phone and adversary (in either order, with any number of other commands in between), such that between these two sets of commands there is no command to corrupt the phone. If S and Zhave non-negligible probability of producing a pair (M, σ) , such that $Ver(pk, M, \sigma)$ accepts, but M was not submitted for signing to \mathcal{F}_{Sig}^{Spl} before the second of the above command sets was issued, then Sig is not UF-CMA secure. *Proof.* We construct a machine $\mathcal{B}^{Sign(sk,\cdot)}(pk)$ that will break the UF-CMA security of Sig. The machine \mathcal{B} executes $\mathcal{F}_{Sig}^{Spl}||\mathcal{Z}||\mathcal{S}$ internally, except:

- Upon initialisation, instead of generating keypair as defined in \mathcal{F}_{Sig}^{Spl} , use key pk that was supplied as input to \mathcal{B} . No sk gets generated by \mathcal{F}_{Sig}^{Spl} . Instead of receiving phone-supplied PIN, \mathcal{F}_{Sig}^{Spl} receives ans stores $\pi(\text{PIN})$ from \mathcal{M}_{PP} . – Upon receiving signing command, instead of computing signature as $\sigma =$
- Upon receiving signing command, instead of computing signature as $\sigma = \text{Sign}(\text{sk}, M)$, make query the the signing oracle that \mathcal{B} has access to and use provided σ to respond signing command.

Suppose that \mathcal{B} outputs a verifiable pair (M, σ) . The message M may have been submitted to the signing oracle, meaning that according to the construction of \mathcal{B} and \mathcal{F}_{Sig}^{Spl} it had to be submitted to \mathcal{F}_{Sig}^{Spl} by either the phone or by \mathcal{S} . By our assumption, there was non-negligible probability that M was either not submitted to \mathcal{F}_{Sig}^{Spl} at all, or it was submitted after the command sets for "Signing by phone and server" and "Signing by adversary and server" were issued by \mathcal{Z} and \mathcal{S} (without intervening commands to increase the corruption level of the phone). In the former case, \mathcal{B} has broken the UF-CMA security of Sig. In the latter case, clone detection mechanism was activated in \mathcal{F}_{Sig}^{Spl} and b_{OK} was set to 0 which means that the adversary was unable to issue signing commands anymore.

4.3 Relation between \mathcal{F}^{gSpl} and \mathcal{F}^{Spl}_{Sig}

The functionalities \mathcal{F}^{gSpl} and \mathcal{F}^{Spl}_{Sig} are obviously related to each other. In fact, the latter can be used to securely implement the former, using protocol Π^{Spl} , given in Fig. 10. The protocol consists of straightforward forwarding of messages between \mathcal{Z} and \mathcal{F}^{Spl}_{Sig} . We argue that the specified handling of corruption requests is reasonable. The adversary could express, how much it wants to corrupt the phone and the server. And the processing is reasonable, considering that the protocol itself is "semi-ideal", i.e. uses a complex ideal functionality.

Theorem 7. If the signature scheme Sig is UF-CMA secure, then protocol Π^{Spl} securely implements \mathcal{F}_{Sig}^{Spl} in \mathcal{F}_{Sig}^{Spl} -hybrid model.

This theorem is proven in App. A.

5 Security of Buldas et al. [6] protocol in UC

In this section, we prove that improved server-supported RSA protocol securely realises \mathcal{F}^{gSpl} in UC model. To achieve it, we model phone's corruptions levels by allowing the adversary to issue commands "*Leak encrypted memory*" and "*Leak unencrypted memory*". When issuing the first command, \mathcal{A} receives the value u. In the case of the second command, \mathcal{A} receives the value d'_1 , as well as u, if this was not sent earlier, which also determines PIN.

```
Protocol \Pi^{Spl}:
```

 Π^{Spl} offers an interface towards \mathcal{Z} for n users: phone, server, and (n-2) entities that only verify signatures.

 Π^{Spl} consists of the machine \mathcal{M}_{ph} , the machine \mathcal{M}_{srv} , and (n-2) machines \mathcal{M}_{gen} . All machines have connections between each other. Additionally, there is the machine \mathcal{F}_{Sig}^{Spl} that is connected to \mathcal{M}_{ph} and \mathcal{M}_{srv} .

Key generation:

- Machine \mathcal{M}_{ph} gets the input (keygen, sid, L, T_0, PIN).
- Machine \mathcal{M}_{srv} gets the input (keygen, sid, T_0). Machine \mathcal{M}_{ph} and machine \mathcal{M}_{srv} both call \mathcal{F}_{Sig}^{Spl} with the respective inputs. They both get back pk. They both send $Ver(pk, \cdot, \cdot)$ to all machines \mathcal{M}_{gen} , where Ver is the verification algorithm of Sig.

Signing: Note that it may happen that only one of \mathcal{M}_{ph} or \mathcal{M}_{srv} gets the command from the environment. The other party's command may come from the adversary, and it would go directly to \mathcal{F}_{Sig}^{Spl} . In this case, \mathcal{F}_{Sig}^{Spl} sends the signature back to \mathcal{A} , too.

- Machine \mathcal{M}_{ph} gets the input (sign, sid, M, PIN) and \mathcal{M}_{srv} gets the input (sign, sid) from the environment.
- They get back (sign_ok, sid) or (sign_fail, sid) or (signature, sid, σ) and pass this back to the environment.

Signature verification: On input (verify, sid, M, σ), run $b \leftarrow Ver(pk, M, \sigma)$ and return b. Corruptions:

- The real adversary \mathcal{A} sends the corruption requests (for either the phone or the server) to \mathcal{F}_{Sig}^{Spl} .
- The latter forwards them either to \mathcal{M}_{ph} or \mathcal{M}_{srv} , which forwards them to the environment.
- The machines \mathcal{M}_{ph} and \mathcal{M}_{srv} do not do anything further with these corruptions.

Fig. 10: Protocol Π^{Spl} securely implementing $\mathcal{F}^{\mathsf{gSpl}}$ in $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ -hybrid model

Theorem 8. If RSA with padding function H is UF-CMA secure, then serversupported RSA signing presented in Figures 4 and 5 is a secure implementation of \mathcal{F}^{gSpl} .

Proof. Let us denote protocols from Figures 4 and 5 as subprotocols of a protocol $\Pi^{4\mathsf{RSASplit}}$. We construct a simulator Sim, such that for any adversary \mathcal{A} , no environment cannot distinguish an execution with $\Pi^{\text{4RSASplit}} \| \mathcal{A}$ from an execution with $\mathcal{F}^{\mathsf{gSpl}} \| Sim \| \mathcal{A}$. Simulator state consists of $\langle \mathsf{c}_{\mathrm{S}}, \mathsf{c}_{\mathrm{P}}, n, n_1, n_2, d_1, d'_1, d''_1, d_2, u, w \rangle$. Next we describe how simulator handles and responds to the commands from F^{gSpl}:

- On command "Corrupt server" from \mathcal{A} before the key generation has started: send (corrupt-server) to \mathcal{F}^{gSpl} .
- On command "Corrupt phone" from \mathcal{A} before the key generation has started: send (corrupt-phone, 3) to \mathcal{F}^{gSpl} .

Universally Composable Server-Supported Signatures for Smartphones 23

- On command (keygen-init) from \mathcal{F}^{gSpl} :
 - If nobody is corrupted: simulate both phone and server. Perform an honest run of the key generation protocol and initialise all the values $n, n_1, n_2, d_1, d'_1, d''_1, d_2, u, w$. The real protocol execution is identical in this case to the simulated one, since all values are generated in the same way as in the real protocol.
 - If the phone is corrupted: simulate the server protocol execution. Generate RSA modulus n_2 , compute private exponent $d_2 = e^{-1} \mod \phi(n_2)$. Upon receiving $\langle n_1, d_1'' \rangle$ from \mathcal{A} , compute $n = n_1 \cdot n_2$, sample clone detection bitstring $w \leftarrow \{0, 1\}^*$ and send $\langle n, w \rangle$ to \mathcal{A} . Set $\mathsf{Ver} = \mathsf{Ver}_{RSA}(\mathsf{pk}, \cdot, \cdot)$ and send (key, Ver) to $\mathcal{F}^{\mathsf{gSpl}}$. Since corrupting phone during key generation corresponds to full corruption $c_{\mathrm{P}} = 3$, receive also (corrupt-pin, PIN) from $\mathcal{F}^{\mathsf{gSpl}}$. The values d_1, d_1', u remain undefined.

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

- If server is corrupted: simulate the phone protocol execution. Generate RSA modulus n_1 , compute private $d_1 = e^{-1} \mod \phi(n_1)$ and sample randomly $d''_1 \leftarrow \mathbb{Z}_{\phi(n_1)}$. 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 $\mathsf{Ver} = \mathsf{Ver}_{RSA}(\mathsf{pk}, \cdot, \cdot)$ and send (key, Ver) to $\mathcal{F}^{\mathsf{gSpl}}$. The values d'_1, d_2, u, n_2 remain undefined. 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 $d'_1 = \mathsf{genShare}(\mathsf{PIN}, u)$. In the simulated, $d''_1 \leftarrow \mathbb{Z}_{\phi(n_1)}$. Since genShare algorithm relies on a PRF which is a bijection, the distribution of the value d''_1 in both cases is identical.
- If both phone and server are 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.
- On command "Corrupt server" from \mathcal{A} : send (corrupt-server) to \mathcal{F}^{gSpl} and proceed as follows:
 - If keys have been generated and phone has not yet been corrupted, then values d_2, d''_1, w, n_1, n_2 have been initialised during key generation. Send d_2, d''_1, w, n_1, n_2 to \mathcal{A} .
 - If keys have been generated and phone has been corrupted, then d_2 has already been defined. Send d_2 to the \mathcal{A} .
- On command "Leak encrypted memory" from \mathcal{A} : ignore this query if keys have not yet been generated. Otherwise, send (corrupt-phone, 1) to \mathcal{F}^{gSpl} and $\langle w, n_1 \rangle$ to the \mathcal{A} and proceed as:
 - If server has not been corrupted: pick a random bitstring $u \leftarrow \{0,1\}^*$ and send it to \mathcal{A} .
 - If server has been corrupted: receive PIN from \mathcal{F}^{gSpl} . Using previously initialised value d'_1 and PIN, compute u by inverting genShare. Send u to the \mathcal{A} .
- On command "Leak unencrypted memory" from \mathcal{A} : ignore this query if keys have not yet been generated. Otherwise, send (corrupt-phone, 2) to \mathcal{F}^{gSpl} and $\langle w, n_1 \rangle$ to the \mathcal{A} . Receive PIN from \mathcal{F}^{gSpl} and proceed as:

- 24 Authors Suppressed Due to Excessive Length
 - If server has not been corrupted: pick a random bitstring u, unless it was already initialised during "Leak encrypted memory". Compute d'_1 from PIN and u using genShare and send $\langle d'_1, u \rangle$ to \mathcal{A} .
 - If server has been corrupted: using previously initialised d'_1 and PIN compute u by inverting genShare. Send $\langle d'_1, u \rangle$ to the \mathcal{A} .
 - On command "Corrupt phone" from \mathcal{A} : send (corrupt-phone, 3) to \mathcal{F}^{gSpl} . If keys have been generated, proceed in the same way as when responding to "Leak unencrypted memory" command.
 - On command "Signing by phone and server": simulate both phone and server, since nobody is corrupted. Perform an honest run of the signing protocol. When signature σ gets initialised, send command (signature, sid, M, σ) to \mathcal{F}^{gSpl} .

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 command "Signing by phone and adversary": simulate phone's protocol execution. Send (sign-server, *sid*) to \mathcal{F}^{gSpl} and get (sign-init, *sid*, M) as response. Calculate signature $\sigma' = H(M)^{d_1 - d''_1} \mod n_1$ and send $\langle \sigma', w \rangle$ to \mathcal{A} . Upon receiving (σ, w) from \mathcal{A} , send command (signature, *sid*, M, σ) to \mathcal{F}^{gSpl} .

The real protocol execution is only different from the simulated one in the way how σ' is calculated. In real protocol signature is calculated as $\sigma' = (M)d'_1$ mod n_1 , where $d'_1 = \text{genShare}(\text{PIN}, u)$. In simulated protocol – $\sigma' = (M)^{d-d''_1}$ mod n_1 . As discussed above, since genShare is a bijection, distribution of d'_1 in the real protocol is identical to $d - d''_1$ in the simulated. Therefore, distribution of σ' is identical in both cases as well.

- On command "Signing by adversary and server": simulate server's protocol execution:
 - If encrypted memory is leaked ($c_P = 1$), simulator has already initialised bitstring's u value. With the first message from \mathcal{A} , receive (σ'_1, w) where $\sigma'_1 = H(M)^{d'_1} \mod n_1$. Having u, test through all possible PIN $\in L$ to find PIN' that produces σ'_1 from $d'_1 = \text{genShare}(u, \text{PIN})$. Set PIN' = PIN and send command (sign, sid, M, PIN') to \mathcal{F}^{gSpl} .
 - If unencrypted memory is leaked $c_P = 2$, simulator has already initialised value u and knows PIN. Set PIN' = PIN and send command (sign, *sid*, M, PIN') to \mathcal{F}^{gSpl} .
 - If phone is fully corrupted $c_{\rm P} = 3$, simulator already knows PIN. Set PIN' = PIN and send command (sign, sid, M, PIN') to \mathcal{F}^{gSpl} .

Upon command (sign-init, sid, M) from \mathcal{F}^{gSpl} , receive (σ'_1, w) from \mathcal{A} . Compute $\sigma_1 = \sigma'_1 \cdot H(M)^{d''_1} \mod n_1$ and $\sigma_2 = H(M)^{d_2} \mod n_2$. Combine σ_1 and σ_2 to the final signature σ using Chinese Remainder Theorem. Verify σ , if it is correct, send command (signature, sid, M, σ) to \mathcal{F}^{gSpl} .

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 command "Verification": signature verification is run which does not involve the simulator.

However, we have to argue that the real protocol (that always executes $Ver(pk, M, \sigma)$ at this point) returns the same verification outcomes as the

ideal functionality (that also uses a table of verification results). We see that due to internal logic of $\mathcal{F}^{\mathsf{gSpl}}$, the only possible deviation is the real protocol returning 1 for some (M, σ) , while the ideal functionality returns 0 because no (M, σ', b) has been recorded for any σ', b and, moreover, $b_{\mathrm{sk}} = 1$. This corresponds to the construction of (M, σ) without M being an argument to some signing query, and without leaking the private key.

This defines the simulator. We see that the composition of the ideal functionality \mathcal{F}^{gSpl} and the simulator *Sim* runs identically to the real protocol $\Pi^{4RSASplit}$ from the point of view of the composition of the environment \mathcal{Z} and the adversary \mathcal{A} , except for the "Verification" commands, where the first may return 0 to \mathcal{Z} while the second returns 1.

We now show that there exist no \mathcal{Z} and \mathcal{A} that can distinguish $\mathcal{F}^{gSpl} \|Sim$ from $\Pi^{4RSASplit}$. Indeed, if there were such \mathcal{Z} and \mathcal{A} , then we can build another adversary \mathcal{B} that receives as an input a bi-prime RSA public key pk and a signing oracle Sign(sk, ·), and breaks the UF-CMA security of RSA. The adversary \mathcal{B} first guesses whether \mathcal{A} is going to leave the phone or the server uncorrupted; this guess is correct with probability at least $\frac{1}{2}$. It then executes the composition $\mathcal{F}^{gSpl} \|\mathcal{Z}\|Sim\|\mathcal{A}$ with the following modifications:

- During the key generation, \mathcal{B} uses pk as the public key share of the to-beleft-uncorrupted party, generating the values of the public key share of the other party.
- When a signature to a message M is requested by \mathcal{F}^{gSpl} from Sim, \mathcal{B} uses both the signing oracle, and the public key share of the to-be-corrupted party to construct that signature, following the construction in the proof of either Theorem 2 or Theorem 3 from Buldas et al [6].
- If \mathcal{Z} makes a verification query (M, σ) to \mathcal{F}^{gSpl} , then check whether it constitutes a forgery in the 4RSASplit scheme. In this case, $\sigma \mod n'$, where n' is the RSA modulus in the public key pk, also constitutes a forged signature of M in RSA scheme, with respect to the key pk. If the forgeries for 4RSASplit occur with non-negligible probability, then \mathcal{B} successfully breaks the security of RSA.

We have seen that the real protocol execution is identical to the simulated one (or, for verification results, at most indistinguishably different), since all values are generated in the same way as in the real protocol. As a result, we see that the views of corrupted phone and server respectively are indistinguishable in the simulation from the real execution.

6 Server-supported ECDSA

In this section, we present a server-supported ECDSA-based protocol between phone and server. We present more illustrative version of key generation and signing protocols in Appendix B.

6.1 Setup

The group \mathbb{G} parameters, generator G and and group order q are predefined to provide security level λ . Hash functions H_0 , H_1 , H_2 are secure implementations of $\mathcal{F}^{\mathsf{rpoRO}}$ with output length 2λ . Parameters for $\mathcal{F}^{\mathsf{MtA}}$ and $\mathcal{F}^{\mathsf{NIZKP}}$ for both phone and server are initialised.

6.2 Key Generation

The phone (client) C starts with generating their share of the secret key x_1 , computing corresponding public key share Q_1 and producing π_{x_1} , a proof of knowledge of x_1 . Phone's next step consists of re-sharing x_1 into x'_1 (that is derived from PIN) and $x''_1 = x_1 - x'_1$ (that will be stored on the server side), which is needed to ensure that the adversary cannot perform offline guessing attacks on user's PIN code. Phone receives user-chosen code PIN and derives key share x'_1 using Algorithm 7 with PIN and a randomly generated bitstring u as input. Phone proceeds by computing corresponding public key share Q'_1 and producing $\pi_{x'_1}$, a proof of knowledge of x'_1 . With the first message, phone commits to values $c_{\text{KG}} = \langle H_1(Q_1, Q'_1, x'', \pi_{x_1}, \pi_{x'_1}) \rangle$.

The server S generates its share of the secret key x_2 , public key Q_2 and produces π_{x_2} , a proof of knowledge of x_2 . The server proceeds with generating clone detection string w. The phone is supposed to send w later during the signing protocol to show to the server that their device memory has not been copied by the adversary. Server sends Q_2, w, π_{x_2} to the phone.

Phone verifies proof π_{x_2} and sends opening to the commitment c_{KG} . Upon receiving opening, server verifies commitment, proofs and that the relation between public key shares holds. Server creates counter T that is needed to track number of incorrect PIN guesses made by the phone.

Finally, both phone and server calculate final public key Q. phone stores only Q, u, w values and must securely delete all the remaining values received or generated during the key generation protocol. This is needed to hide values that adversary can compare their PIN guess against and prevent offline guessing attacks. The formal description of distributed key generation is presented in the Figure 11.

$KeyGen(\lambda)$:

Initialisation:

- a. Based on the given security level λ , generator G and order q is chosen to form a group G.
- b. Threshold of attempts T_0 is defined.
- c. Based on the given security level λ , the parameters for \mathcal{F}^{MtA} and \mathcal{F}^{NIZKP} are initialised for both phone C and server S.
- d. The length of $H_0(\cdot), H_1(\cdot), H_2(\cdot)$ output is defined to be 2λ .

Phone's 1st message:

- a. The phone C generates a random key share $x_1 \leftarrow \mathbb{Z}_q$, calculates a public key share $Q_1 = x_1 \cdot G$ and a proof $\pi_{x_1} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, Q_1, x_1)$.
- b. C takes PIN chosen by the user, generates a random bitstring $u \leftarrow \{0, 1\}^{128}$, calculates $\begin{array}{l} x_1' = \mathsf{genShare}(u,\mathsf{PIN},q) \text{ and } x_1'' = x_1 - x_1' \mod q. \\ \text{c. } C \text{ calculates } Q_1' = x_1' \cdot G \text{ and a proof } \pi_{x_1'} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove},Q_1',x_1'). \end{array}$
- d. C sends $c_{KG} = H_1(Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1})$ to server S.

Server's 1st message:

- a. The server S generates a random key share $x_2 \leftarrow \mathbb{Z}_q$, calculates a public key share $Q_2 = x_2 \cdot G$ and a proof $\pi_{x_2} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, Q_2, x_2)$.
- b. S generates a random bitstring $w \leftarrow \{0,1\}^*$ for the clone detection mechanism. c. S sends $\langle Q_2, \pi_{x_2}, w \rangle$ to C.

Phones's 2nd message:

- a. Upon receiving $\langle Q_2, \pi_{x_2}, w \rangle$, if $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, Q_2, \pi_{x_2}) = 0, C$ sends \perp .
- b. Otherwise, C sends $\langle Q_1, Q_1', x_1'', \pi_{x_1}, \pi_{x_1'} \rangle$.

Server's 2nd message:

- a. S verifies whether $c_{KG} \stackrel{?}{=} H_1(Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1})$, whether $\mathcal{F}^{NIZKP}(\text{verify}, Q_1, \pi_{x_1}) \stackrel{?}{=}$ 0 and $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, Q'_1, \pi_{x'_1}) \stackrel{?}{=} 0$, and also verifies that $Q'_1 + x''_1 \cdot G = Q_1$ holds. If any check failed, then S sends \perp .
- b. Otherwise, S sets counter $T = T_0$.

Output:

- a. C calculates resulting public key $Q = Q_1 + Q_2$, stores $\langle Q, u, w \rangle$ and securely deletes all remaining values received or generated during the key generation protocol.
- b. S calculates $Q = Q_1 + Q_2$ and stores $\langle Q, Q_1, Q_2, Q'_1, x''_1, x_2, w, T \rangle$.

Corruptions during KeyGen:

- If adversary \mathcal{A} corrupts C, it possesses values $x_1, x'_1, x''_1, Q_1, Q_2, u, w$.
- If adversary \mathcal{A} corrupts S, it possesses values x_2, x_1'', Q_1, Q_2, w .

Fig. 11: Distributed key generation ECDSA server-supported protocol

 $\mathbf{Sign}(M, sid)$ Phone's 1st message:

- 1. The phone C generates a random nonce $k_1 \leftarrow \mathbb{Z}_q$, calculates $R_1 = k_1 \cdot G$ and a proof $\pi_{k_1} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_1, k_1)$.
- 2. Upon receiving PIN from a user, C restores secret key share $x'_1 = \text{genShare}(u, \text{PIN}, q)$.
- 3. *C* calculates public key share $Q'_1 = x'_1 \cdot G$ and a proof $\pi_{x'_1} = \mathcal{F}^{\text{NIZKP}}(\text{prove}, Q'_1, x'_1)$.
- 4. C sends $\langle c_{sig} = H_2(R_1, w, M, \pi_{x'_1}, \pi_{k_1}), \pi_{x'_1}, w \rangle$ to S, where M is a message to be signed.

Authenticating phone:

- 1. The server S authenticates the phone C based on response from $\mathcal{F}^{\text{NIZKP}}$: - If $\mathcal{F}^{\text{NIZKP}}(\text{verify}, Q'_1, \pi_{x'_1}) = 1$, C is authenticated.
 - Otherwise, S sends $\langle \hat{Wrong} password, \perp \rangle$ and sets T = T 1.
- 2. Simultaneously, S checks if w sent in the current session by C coincides with their stored copy of w. If not, they halt any further communication with C.

Invoking \mathcal{F}_{MtA} :

- 1. S generates $x_2^* \leftarrow \mathbb{Z}_q$ and calculates $Q_2^* = x_2^* \cdot G$.
- 2. S and C invoke \mathcal{F}_{MtA} functionality with inputs x_2^* and k_1 . They receive corresponding output shares t_c and t_s .

Server's 1st message:

- 1. S generates a masking value $y \leftarrow \mathbb{Z}_q$ and calculates hid $= t_s + x_2^* \cdot y (x_2 + x_1'') \mod q$.
- 2. S generates a random nonce $k_2 \leftarrow \mathbb{Z}_q$, calculates $R_2 = k_2 \cdot G$ and a proof $\pi_{k_2} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_2, k_2)$.
- 3. S sends $\langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2} \rangle$ to C.

Phone's 2nd message:

- 1. C verifies that $(t_c + hid) \cdot G = (y + k_1) \cdot Q_2^* (Q Q_1')$ and, if relation does not hold, C sends \perp .
- 2. If $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, R_{k_2}, \pi_{k_2}) = 0$, then C sends \perp .
- 3. Otherwise, C calculates $R = (y + k_1) \cdot R_2$ and sets $r = r_x \mod q$ such that $R = (r_x, r_y)$.
- 4. C calculates $x_1^* = x_1' (t_c + hid) \mod q$.
- 5. C calculates partial signature $s_1 = (k_1 + y)^{-1}(H_0(M) + rx_1^*) \mod q$ and proof $\pi_{k_1} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_1, k_1).$
- 6. C sends $\langle R_1, s_1, M, \pi_{k_1} \rangle$ to S.

Output:

- 1. S verifies that $c_{sig} = H_2(R_1, w, M, \pi_{x'_1}, \pi_{k_1})$ and if relation does not hold S sends \perp .
- 2. If $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, R_1, \pi_{k_1}) = 0, S \text{ sends } \bot$.
- 3. Otherwise, S calculates $R = k_2 \cdot R_1 + k_2 \cdot y \cdot G$ and assigns $r = r_x \mod q$ such that $R = (r_x, r_y)$.
- 4. S calculates signature value $s = k_2^{-1}(s_1 + r \cdot x_2^*) \mod q$ and verifies signature $\langle r, s \rangle$. If verification failed, S sends \perp .
- 5. S generates new clone detection bitstring $w' \leftarrow \{0, 1\}^*$
- 6. S sends resulting signature $\langle r, s \rangle$ and w' to C.
- 7. C verifies $\langle r, s \rangle$ and sets w = w'.

Corruptions during Sign:

- If adversary \mathcal{A} corrupts S, it possesses values x_2, x_1'', Q_1, Q_2, w .
- If adversary \mathcal{A} get access to client's encrypted memory, it possesses values u, w
- If adversary \mathcal{A} get access to client's unencrypted memory, it possesses values x'_1, u, w

6.3 Signing

With the first message of the signing protocol phone authenticates to the server by sending proof of knowledge of x'_1 , secret key share derived from the user-supplied PIN. Additionally, with the first message, phone sends w that was stored from the previous signing query. In case of abort message \perp sent by the server, the counter is increased T = T + 1. After one successful signature generation, server resets T = 0. In any case, if $T = T_0$ or w sent by phone does not correspond to the value on server's side, server does not participate in any future communication with the phone.

Next, the phone and server run $\mathcal{F}^{\mathsf{MtA}}$ with inputs k_1 and x_2^* to receive correlated values t_c on the phone side and t_s on the server side such that $t_c + t_s = k_1 \cdot x_2^*$. Server proceeds with calculating values for consistency check by generating y, computing hid and sending both values to the phone. Phone checks consistency by verifying that $(t_c + \mathsf{hid}) \cdot G = (y + k_1) \cdot Q_2^* - (Q - Q_1')$. If consistency check passes and proof of knowledge of k_2 verifies, phone calculates partial signature and sends it to the server with π_{k_1} proof of knowledge of k_1 .

If commitment opening and proof π_{k_1} verify, server computes final signature s using partial signature s_1 supplied by the phone. After verifying signature $\langle r, s \rangle$, server generates new clone detection bitstring w' and sends in to the phone. The formal description of our signing protocol is presented in Figure 12.

6.4 Correctness

First, let us show correctness of the check done by the phone during signing in step 5(a). By definition, $Q = Q_1 + Q_2 = Q'_1 + x''_1 \cdot G + Q_2$, $x_1 = x'_1 + x''_1$ and $x_1^* = x'_1 - (t_c + \text{hid})$. Additionally, by definition, $\text{hid} = t_s + x_2^* \cdot y - (x_2 + x''_1)$ and $t_c + t_s = k_1 \cdot x_2^*$. Therefore:

$$\begin{array}{c} (t_c + \mathsf{hid}) \cdot G = (y + k_1) \cdot Q_2^* - (Q - Q_1') \\ (t_c + t_s + x_2^* \cdot y - (x_2 + x_1'') \cdot G = (y + k_1) \cdot Q_2^* - (Q_2 + x_1'' \cdot G) \\ (k_1 \cdot x_2^* + x_2^* \cdot y - (x_2 + x_1'') \cdot G = (y + k_1) \cdot x_2^* \cdot G - (x_2 \cdot G + x_1'' \cdot G) \\ ((y + k_1) \cdot x_2^* - x_2 - x_1'') \cdot G = (y + k_1) \cdot x_2^* \cdot G - (x_2 + x_1'') \cdot G \text{ over } \mathbb{Z}_q \end{array}$$

Now, let us show correctness of the key generation and signing parts of the protocol:

$$\begin{aligned} x_1^* &= x_1' - (t_c + t_s + x_2^* \cdot y - (x_2 + x_1'')). \\ x_1^* &= x_1' - (k_1 \cdot x_2^* + x_2^* \cdot y - (x_2 + x_1'')). \\ x &= x_1' + x_1'' + x_2 = x_1 + x_2 = x_1^* + (k_1 + y) \cdot x_2^* \text{ over } \mathbb{Z}_q. \end{aligned}$$

Let us define $k = k_2 \cdot (k_1 + y)$ and $R = k \cdot G$, then:

By the definition,
$$s = k_2^{-1}(s_1 + r \cdot x_2^*)$$
.
Also, since $s_1 = (k_1 + y)^{-1}(H_0(M) + rx_1^*)$, we have:
 $s = k_2^{-1}[(k_1 + y)^{-1}(H_0(M) + rx_1^*) + rx_2^*] = k_2^{-1}(k_1 + y)^{-1}[H_0(M) + rx_1^* + r(k_1 + y) \cdot x_2^*] =$

$$k^{-1}[H_0(M) + r(x_1^* + (k_1 + y) \cdot x_2^*)] = k^{-1}[H_0(M) + r(x_1' + x_1'' + x_2)] = k^{-1}(H_0(M) + rx) \text{ over } \mathbb{Z}_q$$

Therefore, we have a valid ECDSA signature that can be verified using standard verification algorithm presented in Algorithm 6.

Security of server-supported ECDSA $\mathbf{7}$

In order to show universally composable security of our protocol, we define a simulator Sim, such that for each adversary \mathcal{A} attacking the protocol, the environment \mathcal{Z} cannot distinguish whether it is interacting with the real protocol and the adversary \mathcal{A} , or the ideal functionality \mathcal{F} and the "ideal" adversary which is the composition of \mathcal{A} and Sim. The simulator works by translating messages between the ideal functionality and the real adversary. We show that for all sequences of inputs from \mathcal{Z} and \mathcal{A} , the real protocol proceeds in lock-step with the composition of \mathcal{F} and Sim, with the outputs at each step being indistinguishable.

Theorem 9. Server-supported ECDSA signing presented in Figures 11 and 12 securely implements the functionality $\mathcal{F}_{ECDSA}^{Spl}$ in the $(\mathcal{F}^{NIZKP}, \mathcal{F}^{rpoRO}, \mathcal{F}^{MtA})$ -hybrid model with the presence of a malicious adaptive adversary.

Proof. Let Sig be ECDSA i.e. ECDSA algorithms (Alg. 4-6) are given as parameter to \mathcal{F}_{Sig}^{Spl} . The internal state of the simulator contains the following values, all with the same meaning as in the ideal functionality or in the real protocol. In general, these values are initialized as \perp , and most of them will only get a value when a party has been corrupted. The state of the simulator consists of $c_{\rm P}$, $c_{\rm S}$, pk, $(Q_1, Q'_1, Q''_1, Q_2, x_1, x'_1, x''_1, x_2)$, u, w, b_{OK} , inputs and outputs (or: internal states) of $\mathcal{F}^{\mathsf{NIZKP}}$, $\mathcal{F}^{\mathsf{rpoRO}}$, $\mathcal{F}^{\mathsf{MtA}}$.

Next we describe how simulator handles and responds to the commands from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$:

- On command "Corrupt server" from \mathcal{A} before the key generation has started: send (corrupt-server) to $\mathcal{F}_{ECDSA}^{Spl}$. - On command "Corrupt phone" from \mathcal{A} before the key generation has started:
- send (corrupt-phone, 3) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.
- On command (keygen-start, pk) from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$:
 - If nobody is corrupted: let \mathcal{A} know that the key generation is happening. Simulate key generation protocol (only the presence of messages between the phone and the server), allowing \mathcal{A} to stop it. If \mathcal{A} stops communication, send (keygen-stop) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. If not stopped, then send message (keygen-ok) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and pk to \mathcal{A} .
 - If the phone is corrupted: simulate the server protocol execution. Upon receiving (keygen-start, pk) from $\mathcal{F}_{\text{Sig}}^{\text{Spl}}$ and c_{KG} from \mathcal{A} . Query (observe, \mathcal{A}) to $\mathcal{F}^{\text{rpoRO}}$ and receive set of all queries $\mathcal{Q}_{\mathcal{A}}$ by \mathcal{A} . Find $\langle Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1} \rangle$,

a preimage of c_{KG} . Extract values x_1, x_1'' from proofs $\pi_{x_1}, \pi_{x_1'}$ by running $\mathsf{Extract}(Q_1, \pi_{x_1}, \mathcal{Q}_{\mathcal{A}})$ and $\mathsf{Extract}(Q_1', \pi_{x_1'}, \mathcal{Q}_{\mathcal{A}})$. From $pk = \langle Q, G, q \rangle$ and Q_1 , compute $Q_2 = Q - Q_1$ and simulate a proof π_{x_2} by running $\mathsf{SimProve}(Q_2)$. Sample random bitstring $w \leftarrow \{0,1\}^*$ and send $\langle Q_2, \pi_{x_2}, w \rangle$ to \mathcal{A} . Upon receiving $\langle Q_1, Q_1', x_1'', \pi_{x_1}, \pi_{x_1'} \rangle$ from \mathcal{A} , send (keygen-ok) to $\mathcal{F}_{\mathsf{Spl}}^{\mathsf{Spl}}$.

 $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. One of the differences between the real protocol and the simulation is in the way how values Q_2 and corresponding proof π_{x_2} is generated. In the real protocol, server calculates $Q_2 = x_2 \cdot G$ where $x_2 \leftarrow \mathbb{Z}_q$ and corresponding π_{x_2} by querying $\mathcal{F}^{\mathsf{NIZKP}}$. In the simulation, simulator computes $Q_2 = Q - Q_1$, where Q is provided by $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and $Q_1 = x_1 \cdot G$ using x_1 received through simulator running extraction algorithm. By the way of $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ computing the key Q, we see that the distribution of Q_2 in both cases is identical — it is uniform over \mathbb{G} . Since simulator perfectly simulates the NIZKP, the distribution of π_{x_2} in both cases is also identical.

The other difference, the simulation could fail due to collision or if \mathcal{A} has never queried $\langle Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1} \rangle$ to $\mathcal{F}^{\mathsf{rpoRO}}$. If the server is corrupted: simulate the phone protocol execution. Upon

If the server is corrupted: simulate the phone protocol execution. Upon receiving (keygen-start, pk) from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$, generate a random value c_{KG} and send it to \mathcal{A} . Upon receipt of $\langle Q_2, \pi_{x_2}, w \rangle$, query (observe, \mathcal{A}) to $\mathcal{F}^{\mathsf{rpoRO}}$ to receive set of all queries $\mathcal{Q}_{\mathcal{A}}$ by \mathcal{A} . Learn x_2 by running $\mathsf{Extract}(Q_2, \pi_{x_2}, \mathcal{Q}_{\mathcal{A}})$. Set $Q_1 = Q - Q_2$, pick random x'' and define $Q'_1 = Q_1 - x'' \cdot G$. Simulate proofs $\pi_{x_1}, \pi_{x'_1}$ by running $\mathsf{SimProve}(Q_1)$ and $\mathsf{SimProve}(Q'_1)$. Query (program-RO, $\langle Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1} \rangle$, c_{KG}) to $\mathcal{F}^{\mathsf{rpoRO}}$. Send $\langle Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1} \rangle$ to \mathcal{A} and (keygen-ok) to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. The difference between the real protocol and the simulation is in the way

The difference between the real protocol and the simulation is in the way how values Q_1, Q'_1 and corresponding proofs π_{x_1} and $\pi_{x'_1}$ are generated. In the real protocol, $Q_1 = x_1 \cdot G$ where $x_1 \leftarrow \mathbb{Z}_q$ and in the simulation $Q_1 = Q - Q_2$, where Q is provided by $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and $Q_2 = x_2 \cdot G$ using x_2 received through $\mathcal{F}^{\mathsf{NIZKP}}$. By the way, how $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ computes key Q, we see that the distribution of Q_1 in both cases is identical. In the real protocol $Q'_1 = x'_1 \cdot G$, where $x'_1 = \mathsf{genShare}(u, \mathsf{PIN})$ and in the simulation $Q'_1 = Q_1 - x''_1 \cdot G$ with $x''_1 \leftarrow \mathbb{Z}_q$. Since $\mathsf{genShare}$ relies on PRF which is a bijection function, distribution of value Q'_1 in both cases is identical.

Since simulator perfectly simulates the NIZKP, the distribution of π_{x_1} and $\pi_{x'_1}$ in both cases is also identical. Additionally, there is a negligible probability that programming random oracle fails, due to the collision or the fact that the value $(Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1})$ has been queried to the \mathcal{F}_{rpoRO} before.

- If both phone and the server are fully corrupted: the adversary \mathcal{A} knows full private key sk and can create valid signatures to the messages of its choice. Thus, our scheme does not provide any security guarantees under this setup.
- On command "Corrupt server" from \mathcal{A} : send (corrupt-server) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and proceed as follows:

- If the keys have not yet been generated, then do nothing more.
- If the keys have been generated, and the phone has not yet been corrupted, then select random x₁'', x₂, compute Q₂ = x₂ ⋅ G and Q₁'' = x₁'' ⋅ G. Set Q₁ = Q Q₂ and Q₁' = Q₁ Q₁''. Send ⟨x₁'', x₂, Q₁, Q₂, Q₁⟩ to A.
 If the keys have been generated and the phone has been corrupted, then
- If the keys have been generated and the phone has been corrupted, then the simulator just receives sk from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Since phone has been corrupted, the values x_1, x'_1, x''_1 have already been selected. Define $x_2 = x - x_1$. Send x_2 to the \mathcal{A} . Now the adversary knows the full private key sk and can create valid signatures to the messages of its choice.
- On command "Leak encrypted memory" from \mathcal{A} : ignore this query if keys have not yet been generated. Send (corrupt-phone, 1) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$, w to \mathcal{A} and proceed as follows:
 - If the server has not been corrupted, then pick a random u and it to the \mathcal{A} .
 - If the server has been corrupted, then the simulator just received sk and PIN from $\mathcal{F}_{\text{ECDSA}}^{\text{Spl}}$. The corruption of the server means that the values x_1'' and x_2 have already been selected. Define $x_1' = x_2 x_1''$ and $x_1 = x x_2$. Compute u by inverting genShare() with given PIN and x_1' . Send u to the \mathcal{A} .
- On command "Leak unencrypted memory" from \mathcal{A} : ignore this query if keys have not yet been generated. Send (corrupt-phone, 2) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and get back PIN. Send w to \mathcal{A} and proceed as follows:
 - If the server has not been corrupted, then pick u randomly, unless it has already been picked during a "leak encrypted memory" command. Then compute x'_1 from u and PIN, using genShare(). Send x'_1 and u to \mathcal{A} .
 - If the server has been corrupted, then there has been no "leak encrypted memory" commands coming to the phone. We have already fixed the values x_1'' and x_2 ; together with sk, we can compute x_1' . We use genShare in the backward direction to find u from x_1' and PIN. Send x_1' and u to \mathcal{A} . Now the adversary knows the full private key $\mathsf{sk} = x_2 + x_1' + x_1''$ and can create valid signatures to the messages of its choice. Thus, our scheme does not provide any security guarantees under this setup.
- On command "Corrupt phone" from \mathcal{A} : send (corrupt-phone, 3) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Proceed as follows:
 - If the keys have not yet been generated, then do nothing more.
 - If the keys have been generated, then proceed in the same way as when responding to "leak encrypted memory" command.
- On command "Signing by phone and server": simulate both phone and server, since nobody is corrupted. Perform an honest run of the signing protocol. When signature σ gets initialised, send command (signature, sid, M, σ) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

It is worth to mention the commands (sign-fail) or (sign-success) from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ that indicate an unsuccessful or successful signing of a message. If both parties are uncorrupted, then simulator plays the existence of messages between the phone and the server to \mathcal{A} . If one of the parties is corrupted, then simulator has already been informed beforehand that signing is happening.

- On command "Signing by phone and adversary": simulate phone's protocol execution. Proceed as follows:
 - Run SimProve (Q'_1) to simulate a proof $\pi_{x'}$.
 - Sample random c_{sig} and send $\langle c_{sig}, w, \pi_{x'_1} \rangle$ to \mathcal{A} .
 - Interact with \mathcal{A} on behalf of $\mathcal{F}_{\mathsf{MtA}}$: receive x_2^* as input share and output randomly sampled value t_s .
 - Upon receiving $\langle R_2, Q_2^*, y, \text{hid}, \pi_{k_2} \rangle$ from \mathcal{A} , verify that $\text{hid} = t_s + x_2^* y \cdot (x_2 + x_1'') \mod q$, where x_2, x_1'' are known to the simulator from the key generation process.
 - Send (sign-server, sid) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and get (signature, sid, M, σ) as response, where $\sigma = \langle r, s \rangle$.
 - Calculate a point R from values $\sigma = \langle r, s \rangle$. Query (observe, \mathcal{A}) to $\mathcal{F}^{\mathsf{rpoRO}}$ and receive set of all queries $\mathcal{Q}_{\mathcal{A}}$ by \mathcal{A} . Obtain k_2 by $\mathsf{Extract}(R_2, \pi_{k_2}, \mathcal{Q}_{\mathcal{A}})$. Then compute $R_1 = k_2^{-1} \cdot R y \cdot G$. Query (program-RO, $\langle R_1, w, M, \pi_{x'_1}, \pi_{k_1} \rangle, c_{\operatorname{sig}}$) to $\mathcal{F}^{\mathsf{rpoRO}}$.
 - Compute $s_1 = k_2 \cdot s x_2^* \cdot r$ and run SimProve (k_1) to profuce a proof π_k . Send $\langle R_1, s_1, M, \pi_{x'_1}, \pi_{k_1} \rangle$ to \mathcal{A} . Receive $\langle \sigma, w'$ from \mathcal{A} and set w = w'.

The difference between the real protocol execution and the simulation is in the way how R_1, s_1 are generated. In the real protocol, $R_1 = k_1 \cdot G$, where $k_1 \leftarrow \mathbb{Z}_q$ is sampled by phone. In simulated protocol, $R_1 = k_2^{-1} \cdot R - y \cdot G$, where R is received from $\mathcal{F}_{\text{ECDSA}}^{\text{Spl}}$ as part of signature and $\langle y, k_2 \rangle$ are received from \mathcal{A} . In both cases, R_1 follows uniform distribution over \mathbb{G} . In the real protocol, phone computes $s_1 = (k_1 + y)^{-1}(H_0(M) + r \cdot x_1^*) \mod q$. In simulated protocol, so $s_1 = k_2 \cdot s - x_2^* \cdot r$. Since, we computed $R_1 = k_2^{-1} \cdot R - y \cdot G$ in simulated protocol, it implies $k_1 = k_2^{-1} \cdot k - y \to k_1 + y = k_2^{-1} \cdot k \to (k_1 + y)^{-1} = k_2 \cdot k^{-1}$. Therefore, $s_1 = k_2 \cdot s - x_2^* \cdot r = k_2 \cdot k^{-1}(H(m) + rx) - x_2^* \cdot r = (k_1 + y)^{-1} \cdot (H(m) + rx) - x_2^* \cdot r$. From correctness proof, we know $x = x_1^* + (k_1 + y) \cdot x_2^*$, meaning $(x - x_1^*) \cdot (k_1 + y)^{-1} = x_2^*$. This gives us $s_1 = (k_1 + y)^{-1} \cdot (H(m) + rx) - x_2^* \cdot r = (k_1 + y)^{-1} \cdot (H(m) + rx - r(x - x_1^*)) = (k_1 + y)^{-1} \cdot (H(m) + r \cdot x_1^*) \mod q$, which is the same value as generated in the real protocol. Additionally, there is negligible probability that programming random oracle fails since the value c_{sig} has been queried to the $\mathcal{F}_{\text{rpoRO}}$ already or due to collision. Since simulator perfectly simulates the NIZKPs, the distribution of produced proofs in both cases is also identical.

- On command "Signing by adversary and server": simulate server's protocol execution. Proceed as follows:
 - Receive c_{sig} from \mathcal{A} , query (observe, \mathcal{A}) to \mathcal{F}^{rpoRO} to receive set of all queries $\mathcal{Q}_{\mathcal{A}}$ by \mathcal{A} . Search to find pre-image of c_{sig} to obtain values $\langle R_1, w, M, \pi_{x'_1}, \pi_{k_1} \rangle$.
- Obtain k_1 by $\mathsf{Extract}(R_1, \pi_{k_1}, \mathcal{Q}_{\mathcal{A}})$.
- Interact with \mathcal{A} on behalf of $\mathcal{F}_{\mathsf{MtA}}$: receive k_1 as input share and output randomly sampled value t_c .
- If only encrypted memory is leaked ($c_P = 1$), simulator knows bitstring u. Run Extract($Q'_1 \pi_{x'_1}, Q_A$) to get x'_1 , then test through all possible PINs $\in \{1, \ldots, L\}$ to find PIN' that produces x'_1 from u using genShare(). Proceed with making query (sign-phone, sid, M, PIN') to the $\mathcal{F}_{ECDSA}^{Spl}$. If PIN' is correct, get back (sign-success, sid, M, σ), where $\sigma = \langle r, s \rangle$.

- If unencrypted memory is leaked ($c_P = 2$), simulator knows additionally u and PIN. Set PIN = PIN' and proceed with making query (sign-phone, sid, M, PIN') to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Get back (sign-success, sid, M, σ), where $\sigma = \langle r, s \rangle$.
- If phone is fully corrupted, $(c_P = 2)$, simulator knows x'_1 and PIN. Set $\mathsf{PIN} = \mathsf{PIN}'$ and proceed with making query (sign-success, sid, M, σ) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Get back (sign-success, sid, M, σ), where $\sigma = \langle r, s \rangle$.
- Sample random $y \leftarrow \mathbb{Z}_q$. From $\sigma = \langle r, s \rangle$ compute $R_2 = R \cdot (y + k_1)^{-1}$ and simulate proof π_{k_2} corresponding to R_2 . Sample random hid $\leftarrow \mathbb{Z}_q$ and compute $Q_2^* := (y + k_1)^{-1} \cdot [(t_c + \text{hid}) \cdot G + Q + Q'_1]$. Send $R_2, Q_2^*, y, \text{hid}, \pi_{k_2}$ to \mathcal{A} .
- Upon receiving $\langle R_1, s_1, M, \pi_{x'_1}, \pi_{k_1} \rangle$ from \mathcal{A} , verify that $s_1 \cdot (y \cdot G + R_1) = H(m) \cdot G + r \cdot (x'_1 t_c hid) \cdot G$.
- Generate random $w' \leftarrow \{0,1\}^*$, set w = w' and send $\langle r, s, w' \rangle$ to \mathcal{A} .

The difference between the real protocol execution and simulator is in the way how values R_2 , hid, Q_2^* are generated. In the real protocol, $R_2 = k_2 \cdot G$ with $k_2 \leftarrow \mathbb{Z}_q$ and in the simulated version, $R_2 = R \cdot (y+k_1)^{-1}$ with R being σ received from $\mathcal{F}_{\text{ECDSA}}^{\text{Spl}}$ and $y \leftarrow \mathbb{Z}_q$. Therefore, in both cases, R_2 follows uniform distribution over \mathbb{G} . In the real protocol, hid $= t_s + x_2^* \cdot y - (x_2 + x_1'') \mod q$ with $y, x_2^* \leftarrow \mathbb{Z}_q$ and t_s is received from \mathcal{F}_{MtA} . In the simulated version, hid $\leftarrow \mathbb{Z}_q$. In both cases, hid should be consistent with check $(t_c + \text{hid}) \cdot G = (y+k_1) \cdot Q_2^* - (Q-Q_1')$. If we put in $Q_2^* = (y+k_1)^{-1} \cdot ((t_c + \text{hid}) \cdot G + Q + Q_1')$, we will get $(y+k_1) \cdot (y+k_1)^{-1} \cdot ((t_c+\text{hid}) \cdot G + Q + Q_1') - (Q - Q_1') = (t_c+\text{hid}) \cdot G$, which means that the consistency check successfully passes for simulated hid, Q_2^* . Since simulator perfectly simulates the NIZKP, the distribution of produced proofs in both cases is also identical. Additionally, there is a negligible probability that programming random oracle fails, due to the collision or the fact that the value $(R_1, w, M, \pi_{x_1'}, \pi_{k_1})$ has been queried to the $\mathcal{F}_{\text{rpoRO}}$ before.

We see that the simulator is able to simulate the replies to all the queries made by the adversary, or by the environment. Hence our server-supported ECDSA protocol securely UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

8 Conclusion

In this paper, we introduced the ideal functionalities \mathcal{F}^{gSpl} and \mathcal{F}^{Spl}_{Sig} , which capture the security and functional properties of Smart-ID/SplitKey in general, and for a chosen signature scheme Sig in particular; we have shown that \mathcal{F}^{Spl}_{Sig} can be used to securely implement $\mathcal{F}^{gSpl}_{gSpl}$, if Sig is a secure signature scheme. We have shown that Buldas et al. [6] protocol and our proposed ECDSA protocol securely realise $\mathcal{F}^{gSpl}_{ECDSA}$ correspondingly. The future work could be showing other existing server-supported protocols or modified two-party protocols for other signature schemes (i.e. EdDSA, BLS, ML-DSA) securely realize ideal functionalities \mathcal{F}^{gSpl} and/or \mathcal{F}^{Spl}_{Sig} . **Acknowledgments** This paper is the result of the research project funded by Estonian Research Council under the grant number PRG1780.

References

- W. Diffie, M. Hellman, New directions in cryptography, IEEE Transactions on Information Theory 22 (6) (1976) 644–654. doi:10.1109/TIT.1976.1055638.
- Y. Desmedt, Society and group oriented cryptography: a new concept, in: C. Pomerance (Ed.), Advances in Cryptology — CRYPTO '87, Springer Berlin Heidelberg, Berlin, Heidelberg, 1988, pp. 120–127.
- 3. L. Brandão, R. Peralta, Nist first call for multi-party threshold schemes (2023).
- 4. L. Brandão, M. Davidson, A. Vassilev, Nist roadmap toward criteria for threshold schemes for cryptographic primitives (2020).
- P. Laud, J. Vakarjuk, A comparison-based methodology for the security assurance of novel systems, in: S. K. Katsikas, F. Cuppens, C. Kalloniatis, J. Mylopoulos, F. Pallas, J. Pohle, M. A. Sasse, H. Abie, S. Ranise, L. Verderame, E. Cambiaso, J. M. Vidal, M. A. S. Monge, M. Albanese, B. Katt, S. Pirbhulal, A. Shukla (Eds.), Computer Security. ESORICS 2022 International Workshops - CyberICPS 2022, SECPRE 2022, SPOSE 2022, CPS4CIP 2022, CDT&SECOMANE 2022, EIS 2022, and SecAssure 2022, Copenhagen, Denmark, September 26-30, 2022, Revised Selected Papers, Vol. 13785 of Lecture Notes in Computer Science, Springer, 2022, pp. 625–644. doi:10.1007/978-3-031-25460-4_36.
- A. Buldas, A. Kalu, P. Laud, M. Oruaas, Server-supported rsa signatures for mobile devices, in: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I, Springer, 2017, pp. 315–333.
- 7. J. Jonsson, B. Kaliski, Rfc3447: Public-key cryptography standards (pkcs) №1: Rsa cryptography specifications version 2.1, Tech. rep., RFC Editor, USA (2003).
- A. P. Sarr, Cryptanalysis and improvement of smart-id's clone detection mechanism, Cryptology ePrint Archive, Paper 2019/1412 (2019). URL https://eprint.iacr.org/2019/1412
- R. Canetti, Universally composable security: A new paradigm for cryptographic protocols, in: Proceedings 42nd IEEE Symposium on Foundations of Computer Science, IEEE, 2001, pp. 136–145.
- A. K. Lenstra, Key length. contribution to the handbook of information security (2004).
- M. J. Hinek, On the security of multi-prime rsa, Journal of Mathematical Cryptology 2 (2) (2008) 117–147.
- P. Shor, Algorithms for quantum computation: discrete logarithms and factoring, in: Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124–134. doi:10.1109/SFCS.1994.365700.
- D. Stebila, S. Fluhrer, S. Gueron, Hybrid key exchange in TLS 1.3, Internet-draft, Internet Engineering Task Force (Sep. 2023). URL https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/09/
- N. Bindel, U. Herath, M. McKague, D. Stebila, Transitioning to a quantum-resistant public key infrastructure, in: Post-Quantum Cryptography: 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings 8, Springer, 2017, pp. 384–405.

- 36 Authors Suppressed Due to Excessive Length
- A. A. Giron, R. Custódio, F. Rodríguez-Henríquez, Post-quantum hybrid key exchange: a systematic mapping study, Journal of Cryptographic Engineering 13 (1) (2023) 71–88. doi:10.1007/s13389-022-00288-9.
- D. Moody, Module-lattice-based digital signature standard, Tech. rep., National Institute of Standards and Technology (2023). doi:10.6028/nist.fips.204.ipd.
- D. Moody, Stateless hash-based digital signature standard, Tech. rep., National Institute of Standards and Technology (2023). doi:10.6028/nist.fips.205.ipd.
- T. Rabin, A simplified approach to threshold and proactive rsa, in: H. Krawczyk (Ed.), Advances in Cryptology — CRYPTO '98, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 89–104.
- V. Shoup, Practical threshold signatures, in: B. Preneel (Ed.), Advances in Cryptology — EUROCRYPT 2000, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 207–220.
- D. Boneh, M. Franklin, Efficient generation of shared rsa keys, J. ACM 48 (4) (2001) 702-722. doi:10.1145/502090.502094. URL https://doi.org/10.1145/502090.502094
- C. Hazay, G. L. Mikkelsen, T. Rabin, T. Toft, Efficient rsa key generation and threshold paillier in the two-party setting, in: O. Dunkelman (Ed.), Topics in Cryptology – CT-RSA 2012, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 313–331.
- M. Chen, J. Doerner, Y. Kondi, E. Lee, S. Rosefield, A. Shelat, R. Cohen, Multiparty generation of an rsa modulus, Journal of Cryptology 35 (2) (2022) 12. doi: 10.1007/s00145-021-09395-y. URL https://doi.org/10.1007/s00145-021-09395-y
- D. Boneh, J. Horwitz, Generating a product of three primes with an unknown factorization, in: J. P. Buhler (Ed.), Algorithmic Number Theory, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 237–251.
- I. Damgård, G. L. Mikkelsen, T. Skeltved, On the security of distributed multiprime rsa, in: J. Lee, J. Kim (Eds.), Information Security and Cryptology - ICISC 2014, Springer International Publishing, Cham, 2015, pp. 18–33.
- J. Camenisch, A. Lehmann, G. Neven, K. Samelin, Virtual smart cards: How to sign with a password and a server, in: V. Zikas, R. De Prisco (Eds.), Security and Cryptography for Networks, Springer International Publishing, Cham, 2016, pp. 353–371.
- P. MacKenzie, M. K. Reiter, Two-party generation of dsa signatures, in: J. Kilian (Ed.), Advances in Cryptology — CRYPTO 2001, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 137–154.
- Y. Lindell, Fast secure two-party ecdsa signing, in: Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017, Proceedings, Part II 37, Springer, 2017, pp. 613–644.
- 28. Y. Lindell, Fast secure two-party ecdsa signing, Journal of Cryptology 34 (2021) 1–38.
- J. Doerner, Y. Kondi, E. Lee, A. Shelat, Secure two-party threshold ecdsa from ecdsa assumptions, in: 2018 IEEE Symposium on Security and Privacy (SP), IEEE, 2018, pp. 980–997.
- G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, I. Tucker, Two-party ecdsa from hash proof systems and efficient instantiations, in: A. Boldyreva, D. Micciancio (Eds.), Advances in Cryptology – CRYPTO 2019, Springer International Publishing, Cham, 2019, pp. 191–221.
- 31. A. Dalskov, C. Orlandi, M. Keller, K. Shrishak, H. Shulman, Securing dnssec keys via threshold ecdsa from generic mpc, in: L. Chen, N. Li, K. Liang, S. Schneider (Eds.),

Computer Security – ESORICS 2020, Springer International Publishing, Cham, 2020, pp. 654–673.

- 32. H. Xue, M. H. Au, X. Xie, T. H. Yuen, H. Cui, Efficient online-friendly two-party ecdsa signature, in: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 558–573. doi:10.1145/3460120.3484803.
- S. Kocaman, Y. Talibi Alaoui, Efficient secure two party ecdsa, in: E. A. Quaglia (Ed.), Cryptography and Coding, Springer Nature Switzerland, Cham, 2024, pp. 161–180.
- R. Gennaro, S. Goldfeder, A. Narayanan, Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security, in: M. Manulis, A.-R. Sadeghi, S. Schneider (Eds.), Applied Cryptography and Network Security, Springer International Publishing, Cham, 2016, pp. 156–174.
- 35. Y. Lindell, A. Nof, Fast secure multiparty ecdsa with practical distributed key generation and applications to cryptocurrency custody, in: In the 2018 ACM SIGSAC Conference, 2018, pp. 1837–1854. doi:10.1145/3243734.3243788.
- 36. R. Gennaro, S. Goldfeder, Fast multiparty threshold ecdsa with fast trustless setup, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 1179–1194. doi:10.1145/3243734.3243859.
- J. Doerner, Y. Kondi, E. Lee, A. Shelat, Threshold ecdsa from ecdsa assumptions: The multiparty case, in: 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 1051–1066. doi:10.1109/SP.2019.00024.
- G. Castagnos, D. Catalano, F. Laguillaumie, F. Savasta, I. Tucker, Bandwidth-efficient threshold ec-dsa, in: A. Kiayias, M. Kohlweiss, P. Wallden, V. Zikas (Eds.), Public-Key Cryptography – PKC 2020, Springer International Publishing, Cham, 2020, pp. 266–296.
- R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, U. Peled, Uc non-interactive, proactive, threshold ecdsa with identifiable aborts, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, CCS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 1769–1787. doi:10.1145/3372297.3423367. URL https://doi.org/10.1145/3372297.3423367
- Y. Deng, S. Ma, X. Zhang, H. Wang, X. Song, X. Xie, Promise sigma-protocol: How to construct efficient threshold ecdsa from encryptions based on class groups, in: M. Tibouchi, H. Wang (Eds.), Advances in Cryptology – ASIACRYPT 2021, Springer International Publishing, Cham, 2021, pp. 557–586.
- M. Pettit, Efficient threshold-optimal ecdsa, in: M. Conti, M. Stevens, S. Krenn (Eds.), Cryptology and Network Security, Springer International Publishing, Cham, 2021, pp. 116–135.
- 42. J. Doerner, Y. Kondi, E. Lee, A. Shelat, Threshold ecdsa in three rounds, in: 2024 IEEE Symposium on Security and Privacy (SP), 2024, pp. 3053–3071. doi:10.1109/SP54263.2024.00178.
- 43. R. Canetti, Universally composable security, J. ACM 67 (5) (Sep. 2020). doi: 10.1145/3402457. URL https://doi.org/10.1145/3402457
- 44. D. Johnson, A. Menezes, S. Vanstone, The elliptic curve digital signature algorithm (ecdsa), International journal of information security 1 (2001) 36–63.
- 45. M. Bellare, P. Rogaway, The exact security of digital signatures how to sign with RSA and rabin, in: U. M. Maurer (Ed.), Advances in Cryptology - EUROCRYPT

'96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding, Vol. 1070 of Lecture Notes in Computer Science, Springer, 1996, pp. 399–416. doi:10.1007/3-540-68339-9_34.

- 46. M. Bellare, P. Rogaway, Random oracles are practical: A paradigm for designing efficient protocols, in: Proceedings of the 1st ACM Conference on Computer and Communications Security, 1993, pp. 62–73.
- 47. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, G. Neven, The wonderful world of global random oracles, in: J. B. Nielsen, V. Rijmen (Eds.), Advances in Cryptology – EUROCRYPT 2018, Springer International Publishing, Cham, 2018, pp. 280–312.
- C. P. Schnorr, Efficient identification and signatures for smart cards, in: G. Brassard (Ed.), Advances in Cryptology — CRYPTO' 89 Proceedings, Springer New York, New York, NY, 1990, pp. 239–252.
- A. Fiat, A. Shamir, How to prove yourself: Practical solutions to identification and signature problems, in: A. M. Odlyzko (Ed.), Advances in Cryptology — CRYPTO' 86, Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, pp. 186–194.
- 50. A. Lysyanskaya, L. N. Rosenbloom, Universally composable σ-protocols in the global random-oracle model, in: E. Kiltz, V. Vaikuntanathan (Eds.), Theory of Cryptography, Springer Nature Switzerland, Cham, 2022, pp. 203–233.
- M. Fischlin, Communication-efficient non-interactive proofs of knowledge with online extractors, in: V. Shoup (Ed.), Advances in Cryptology – CRYPTO 2005, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 152–168.
- Y.-H. Chen, Y. Lindell, Optimizing and implementing fischlin's transform for UC-secure zero knowledge, IACR Communications in Cryptology 1 (2) (2024). doi:10.62056/a66chey6b.
- M. Keller, E. Orsini, P. Scholl, Actively secure of extension with optimal overhead, in: R. Gennaro, M. Robshaw (Eds.), Advances in Cryptology – CRYPTO 2015, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 724–741.
- L. Roy, Softspokenot: Quieter of extension from small-field silent vole in the minicrypt model, in: Annual International Cryptology Conference, Springer, 2022, pp. 657–687.
- P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: J. Stern (Ed.), Advances in Cryptology — EUROCRYPT '99, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 223–238.
- G. Castagnos, F. Laguillaumie, Linearly homomorphic encryption from ddh, in: K. Nyberg (Ed.), Topics in Cryptology — CT-RSA 2015, Springer International Publishing, Cham, 2015, pp. 487–505.
- 57. H. Xue, M. H. Au, M. Liu, K. Y. Chan, H. Cui, X. Xie, T. H. Yuen, C. Zhang, Efficient multiplicative-to-additive function from joye-libert cryptosystem and its application to threshold ecdsa, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 2974–2988. doi:10.1145/3576915.3616595. URL https://doi.org/10.1145/3576915.3616595

All links were last followed on November 29, 2024.

A Implementing $\mathcal{F}^{\mathsf{gSpl}}$ using $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$

In this section, we prove Theorem 7. We construct a simulator Sim that translates between the messages from \mathcal{F}^{gSpl} and the messages from \mathcal{A} . Internally, Simsimulates \mathcal{F}^{Spl}_{Sig} . Next, we describe the way how the simulator Sim responds to the commands:

<u>Key generation</u>: receive (keygen) from \mathcal{F}^{gSpl} . Simulate \mathcal{F}^{Spl}_{Sig} , including the communications with \mathcal{A} . During this, *Sim* gets (pk, sk) generated (even if either phone or server is corrupt). Sends (key, Ver(pk, \cdot, \cdot)) to \mathcal{F}^{gSpl} .

Signing:

- Request from phone and server: in this case, the clone detection checks and the PIN correctness checks are done by \mathcal{F}^{gSpl} . While \mathcal{F}^{gSpl} is processing the signing query, it makes a signature request for message M to Sim. Compute $\sigma \leftarrow \text{s} \text{Sign}(\text{sk}, M)$ and returns it to \mathcal{F}^{gSpl} .
- Request from phone and adversary: in this case, the environment expects to make the (sign, sid, M, PIN') request to \mathcal{M}_{ph} , while adversary expects to make (sign, sid) request to \mathcal{F}_{Sig}^{Spl} . The former request is actually received by \mathcal{F}^{gSpl} , while the latter request is received by Sim. Pass that request to \mathcal{F}^{gSpl} , too. Now, \mathcal{F}^{gSpl} again does the checks and requests a signature for message M. Computes $\sigma \leftarrow \text{s} Sign(sk, M)$ and returns σ to \mathcal{F}^{gSpl} . He also sends σ to \mathcal{A} , because it is expecting it, too.
- Request from adversary and phone: in this case, the environment expects to make the (sign, sid) query to \mathcal{M}_{srv} while \mathcal{A} expects to make the (sign, sid, M, PIN') query to \mathcal{F}^{Sig} . Again, the former request is actually received by \mathcal{F}^{gSpl} , while the latter request is received by Sim. Pass this request to \mathcal{F}^{gSpl} , too. The computations and message exchanges continue as above. If \mathcal{F}^{gSpl} requests for a signature for the message M, then construct σ and give it to both \mathcal{F}^{gSpl} and to \mathcal{A} . If the signature request actually takes place, then corruption level $c_{\rm P}$ is set to be ≥ 2 . The first time it happens, \mathcal{F}^{gSpl} sends PIN to the ideal adversary, i.e. to Sim. Forward it to \mathcal{A} , because \mathcal{A} expects to receive PIN from \mathcal{F}^{Sig} .

Corruptions

The adversary \mathcal{A} makes corruption requests to \mathcal{F}_{Sig}^{Spl} , i.e. they reach *Sim*. The simulator forwards them to \mathcal{F}^{gSpl} . If \mathcal{F}^{gSpl} sends back PIN, then *Sim* forwards it to \mathcal{A} . If, according to the internal logic of \mathcal{F}_{Sig}^{Spl} , the private key sk should be sent to the adversary, then *Sim* does it.

Verification

A user makes a verification request for (M, σ) that is received by \mathcal{F}^{gSpl} . The functionality \mathcal{F}^{gSpl} may decide on the result by itself, or it may forward the verification request to the simulator. The simulator then answers with $Ver(pk, M, \sigma)$.

Security of verification follows from the UF-CMA security of Sig. As long as one of phone and server is uncorrupted, the adversary does not have access to sk. Hence if \mathcal{M}_{gen} would return that a signature verifies, but there is no

 (M, σ, b) recorded in the database of \mathcal{F}^{gSpl} , we have a forgery. In this case we could turn the simulator, \mathcal{A} , and \mathcal{Z} to a challenger that breaks the UF-CMA security of Sig. The challenger would work exactly like $\mathcal{Z}||Sim||\mathcal{A}$ together, but instead of generating (pk, sk) it would use pk and the signing oracle given by the environment of the UF-CMA experiment. There will be no need of producing sk, as long as one of the parties is uncorrupted.

B Server-supported ECDSA protocols diagrams

In this section, we provide more illustrative presentation of two-party key generation and signing protocols.

Key generation	
Phone	Server
$x_1 \leftarrow \mathbb{Z}_q, Q_1 = x_1 \cdot G$	$x_2 \leftarrow \mathbb{S} \mathbb{Z}_q, Q_2 = x_2 \cdot G$
$\pi_{x_1} = \mathcal{F}^{NIZKP}(prove, Q_1, x_1)$	$\pi_{x_2} = \mathcal{F}^{NIZKP}(prove, Q_2, x_2)$
Get PIN from user; $u \leftarrow \{0, 1\}$	$w \leftarrow \{0,1\}^*$
$x_1' = genShare(u,PIN,q)$	
$x_1'' = x_1 - x_1' \mod q, Q_1' = x_1' \cdot G$	
$\pi_{x_1'} = \mathcal{F}^{NIZKP}(prove,Q_1',x_1')$	
	$\langle c_{\mathrm{KG}} = H_1(Q_1, Q_1', x_1'', \pi_{x_1}, \pi_{x_1'}) \rangle$

 $\langle Q_2, \pi_{x_2}, w \rangle$

If $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, Q_2, \pi_{x_2}) = 0$, return \bot

open =
$$\langle Q_1, Q'_1, x''_1, \pi_{x_1}, \pi_{x'_1} \rangle$$

	If $c_{KG} \neq H_1(open)$, return \perp
	If $Q'_1 + x''_1 \cdot G \neq Q_1$, return \perp
	If $\mathcal{F}^{NIZKP}(verify, Q_1, \pi_{x_1}) = 0$, return \perp
	If $\mathcal{F}^{NIZKP}(verify,Q_1',\pi_{x_1'})=0$, return \perp
	Initiate the PIN attempt counter T
$Q = Q_1 + Q_2$	$Q = Q_1 + Q_2$
Store $\langle Q, u, w \rangle$	Store $\langle Q, Q_1, Q_2, Q_1', x_1'', x_2, w, T \rangle$

Signing

Phone Server $k_1 \leftarrow \mathbb{Z}_q, R_1 = k_1 \cdot G$ Get PIN from user $x'_1 = \mathsf{genShare}(u, \mathsf{PIN}, q)$ $Q_1' = x_1' \cdot G$ $\pi_{x_1'} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, Q_1', x_1')$ $\pi_{k_1} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_1, k_1)$ $\langle c_{\rm sig}, w, \pi_{x_1'} \rangle$ $c_{sig} = H_2(R_1, w, M, \pi_{x'_1}, \pi_{k_1})$ Verify w. If failed, cease communication. If $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify},Q_1',\pi_{x_1'})=0, T=T+1$ $x_2^* \leftarrow \mathbb{Z}_q, Q_2^* = x_2^* \cdot G$ Input: k_1 Input: x_2^* $Output: t_c$ $Output: t_s$ $y \leftarrow \mathbb{Z}_q$, hid $= t_s + x_2^* \cdot y - (x_2 + x_1'') \mod q$ $k_2 \leftarrow \mathbb{Z}_q, R_2 = k_2 \cdot G$ $\langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2} \rangle$ $\pi_{k_2} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_2, k_2)$ If $(t_c + hid) \cdot G \neq (y + k_1) \cdot Q_2^* - (Q - Q_1')$, return \perp $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, R_{k_2}, \pi_{k_2}) = 0, \text{ return } \bot$ $R = (r_x, r_y) = (y + k_1) \cdot R_2$ $r = r_x \mod q$ $x_1^* = x_1' - (t_c + \mathsf{hid}) \mod q$ $s_1 = (k_1 + y)^{-1} (H_0(M) + rx_1^*) \mod q$ $\xrightarrow{\langle R_1, s_1, M, \pi_{x_1'}, \pi_{k_1} \rangle}$ $\pi_{k_1} = \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_1, k_1)$ open = $(R_1, w, M, \pi_{x'_1}, \pi_{k_1})$ If $c_{sig} \neq H_2(open)$, return \perp If $\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, R_1, \pi_{k_1}) = 0$, return \bot $R = (r_x, r_y) = k_2 \cdot R_1 + k_2 \cdot y \cdot G$ $r = r_x \mod q$ $s = k_2^{-1}(s_1 + r \cdot x_2^*) \mod q$ If $Ver(M, \langle r, s \rangle, pk) = 0$, return \bot $\langle r, s, w' \rangle$ $w' \leftarrow \{0, 1\}^*$ If $\operatorname{Ver}(M, \langle r, s \rangle, pk) = 0$, return \perp Set w := w'

Store w'