Universally Composable Server-Supported Signatures for Smartphones

Nikita Snetkov[1,2], Jelizaveta Vakarjuk[1,2] and Peeter Laud[1]

[1] Cybernetica AS, Tallinn, Estonia
[2] Tallinn University of Technology, Tallinn, Estonia

Abstract.
Smart-ID is a signing and authentication service available to residents of Belgium, Estonia, Iceland, Latvia and Lithuania. Such smartphone-based signing service delivers the usability of software keys with security guarantees of ID-cards by relying on a remote server to compensate for weak device protection. Security of current Smart-ID relies on multi-prime server-supported RSA, password-authenticated key shares and clone detection mechanism. Unfortunately, the security properties of the underlying protocol (Buldas et al. ESORICS 2017) have been specified only in "game-based" manner. In other words, there are no guarantees for the protocol being secure in concurrent setting. We remedy this shortcoming by presenting two ideal functionalities $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ for server-supported signing in Universal Composability model. We show that improved RSA server-supported protocol realizes $\mathcal{F}^{\mathsf{gSpl}}$ and give practical example of server-supported ECDSA protocol realizing $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$.

Keywords: universal composability · server-supported signatures

1 Introduction

For nearly half a century, digital signatures have been a foundational element in securing computer systems, providing essential security properties such as data integrity, authenticity, and non-repudiation. Their widespread adoption could be illustrated by the usage in cryptographic protocols such as PGP, SSH, TLS and distributed ledgers. The legal recognition of digital signatures has been formalized by eIDAS in the European Union (EU), ESIGN in the United States, and ZertES in Switzerland, where they are granted the same legal status as handwritten signatures.

Given the extensive deployment of digital signatures, necessity occurs for robust cryptographic key protection mechanisms. While there are solutions such as Trusted Execution Environments (TEE), Hardware Security Modules (HSMs) and smart cards, they do not offer much flexibility or scalability. Meanwhile, software-based solutions could be more conveniently deployed and updated across various platforms, including desktop computers, smartphones and smart devices. Nevertheless, software-only solutions could introduce vulnerabilities in the management of private keys.

Security issues of software-only solutions deployed on partially secure platforms could be mitigated by introducing dedicated servers for remote signing. However, such infrastructure could reduce the control that individuals have over their private keys. The issues of both security and ownership may be mitigated by applying multiparty/threshold cryptography [Des88, Lin20]. Smart-ID [1] is an example of a deployed solution utilizing supporting

E-mail: nikita.snetkov@cyber.ee (Nikita Snetkov), jelizaveta.vakarjuk@cyber.ee (Jelizaveta Vakarjuk), peeter.laud@cyber.ee (Peeter Laud)

[1] www.smart-id.com

This work is licensed under a "CC BY 4.0" license.

servers and distributed signing. It provides an authentication and digital signature creation services for over three million users in Belgium, Estonia, Iceland, Latvia, and Lithuania. The cryptographic protocol behind Smart-ID, *server-supported signing* [BKLO17], allows distribution of a private key between the user's smartphone and a dedicated server. Smart-ID's server-supported signing is built upon RSA 2-out-of-2 threshold signing with protection mechanisms against offline dictionary attack and memory cloning. The public keys and the signatures created via Smart-ID are indistinguishable from standard RSA signatures and public keys. This makes Smart-ID *functionally interchangeable* [Bra26] with RSA and interoperable with existing protocols/applications. It could be even argued that Smart-ID is a no less secure solution than traditional smart card-based approach [LV22]. Moreover, Smart-ID is a Qualified Signature Creation Device (QSCD), i.e. Smart-ID signatures are legally equivalent to the handwritten signatures in the European Union (EU) [2].

1.1 Motivation

The server-supported signing behind Smart-ID was proposed by Buldas et al. [BKLO17]. Their protocol enables a client using a constrained device (i.e. phone, smartwatch) and a supporting server collaboratively produce multi-prime RSA signature [Hin08]. The protocol design takes into account differences in capabilities of clients and servers, offering multiple techniques to mitigate risks across varying degrees of client's device compromise. This solution is effective and practical, but its security analysis could be improved. Buldas et al. define 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 [Sar19] against the protection mechanism which catches malicious copies of client's memory (fortunately, that attack has been mitigated in the deployed service[3]). Buldas et al. [BKLO17] 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 defining the system behavior in the ideal world is necessary. The Smart-ID security should be analyzed in a framework which allows to discuss security of protocols in concurrent composition, for example, in the universal composability (UC) framework [Can01, Can20].

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

1. *Identity wallets*: recently updated version of the European Digital Identity Framework Regulation i.e. eIDAS2 requires EU's Member States to develop an EU Digital Identity wallet (EUDI wallet) solution available to their residents and public/private organizations [4]. Such wallets aimed to provide its users a digital identity storage and privacy-preserving authentication services. The eIDAS2 regulation requires seamless interoperability for EUDI wallets, resulting in the development of an architecture and reference framework (ARF) [Eur25]. It defines common attribute formats, standards, technical specifications, and guidelines to enable consistent implementation of the eIDAS2. With regard to signature schemes, the majority of EUDI Wallet ARF's attribute formats are aligned with ECDSA, not RSA [ISO21, SLCH25].

2. *Migration to quantum safe-solutions*: with National Institute of Standards and

[2] https://www.smart-id.com/e-service-providers/smart-id-as-a-qscd/

[3] https://www.tuvit.de/fileadmin/Content/TUV_IT/zertifikate/en/9263BE_s.pdf

[4] https://ec.europa.eu/digital-building-blocks/sites/spaces/EUDIGITALIDENTITYWALLET/pages/694487738/EU+Digital+Identity+Wallet+Home

Technology (NIST) releasing a set of new quantum-safe standards [5], and overall global trend in adoption of post-quantum cryptography, substitution of RSA signing with schemes like ML-DSA, Falcon or SLH-DSA is becoming inevitable. Though, migration of authentication and signing solutions is less urgent than for encryption ones [6], there remains a pressing need to develop post-quantum server-supported signing solutions, given the risk that quantum threats could mature more quickly than expected, necessitating proactive readiness.

3. *Hybrid signatures for quantum-safe solutions*: To provide proper migration to quantum-secure schemes, one of the strategies is support hybrid modes [SFG23, BHMS17, GCRH23], i.e. using both quantum-safe and quantum-vulnerable algorithms. This strategy could prevent from any potential vulnerabilities exploits in pure post-quantum solutions. Due to the large sizes of post-quantum signatures standardised by NIST, it makes sense to employ post-quantum signatures together with ECDSA/EdDSA rather than RSA signatures during a transition period. German Federal Office for Information Security (BSI) is already following this approach and ceases the issuance of RSA certificates starting from 2029 while using ECDSA in hybrid solutions [7].

Therefore, there is a need to define security of Smart-ID (i.e. server-supported signatures) to be parametrized by any signature scheme, while keeping its core security properties intact.

1.2 Our contribution

1. We propose an ideal functionality $\mathcal{F}^{\mathsf{gSpl}}$ (Section 3.1) that captures adaptive security and operational behavior of a server-supported signatures consistent with the design of Smart-ID. We define an additional ideal functionality $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ (Section 3.2) which can be parameterized by any signature scheme Sig. Functionality $\mathcal{F}^{\mathsf{gSpl}}$ may be seen as *generic signing service*, while $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ is a server-supported signing functionality with concrete signature scheme as a parameter.

2. We present a UC-secure construction of improved Buldas et al. [BKLO17] server-supported protocol adapted to resolve the issues raised by Sarr [Sar19] that securely realizes $\mathcal{F}^{\mathsf{gSpl}}$ functionality (Section 4).

3. To show capabilities of $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$, we show that a UC-variant of Xue et al. two-party ECDSA protocol [XAX+21] securely realizes $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ (Section 6).

1.3 Related Work

The expression *server-supported signatures*, originally coined by Asokan et al. [ATW97], has become ambiguous and interchangeable with the *server-assisted* and *server-aided* terms. By above mentioned definitions Asokan et al. [ATW97] and sequential works [MR01b, BB04, BB05, GT12] implied protocols with server assuming responsibility for the computationally demanding signing process, minimizing the processing overhead on the client side. In this work, we are interested in server-supported protocols built upon threshold signing [Des88] and multisignatures [Oka88]. Buldas et al. [BKLO17] protocol, which is used as foundation for Smart-ID, falls into this category. Several works [Gan95, MR01a, XS03, DEF24] have studied systems with similar properties to Smart-ID, but none provide analysis within

[5] https://blog.cloudflare.com/nists-first-post-quantum-standards/

[6] https://pqcrypto2025.iis.sinica.edu.tw/slides/Invited2.pdf

[7] https://docbox.etsi.org/Workshop/2023/02_QUANTUMSAFECRYPTOGRAPHY/TECHNICALTRACK/WORLDTOUR/BSI_KOUSIDIS.pdf

the UC model, except for the work of Camenish et al. [CLNS16]. While their protocol Pass2Sign offers UC password-authenticated server-supported signing, it lacks distributed key generation and clone detection mechanisms present in Smart-ID.

2 Preliminaries

2.1 General Notation

The relation $a \leftarrow\!\$\ A$ denotes sampling an element a uniformly at random from the set A. We use $:=$ for defining a new value in a deterministic manner. $\mathbb{Z}_q$ – denotes a set of integers modulo q, Symbol $\mathbb{N}$ - set of naturals, $\mathbb{G}$ – a group, $\mathbb{F}$ – a finite field, $\mathbb{P}(\lambda)$ – a set of primes of size defined by λ. The symbol $\perp$ is used to indicate a failure or abort. The symbol $\|$ means depending on the context: a) parallel execution of Turing machines or protocols b) concatenation of two inputs. We use $\phi(\cdot)$ to denote Euler's totient function.

2.2 Universal Composability

The Universal Composability (UC) framework, introduced by Canetti [Can01, Can20], defines when a system Π composed of interactive Turing machines securely implements another such system. The system may consist of several interacting machines $\mathcal{M}_i$ executing the functionality of some protocol, and also with machines $\mathcal{F}_{\mathsf{fun}}$ executing some ideal functionality, e.g. the random oracle. Or system may consist of a single machine $\mathcal{F}$ that states what the *ideal functionality* of some cryptographic construction is. The system Π is not closed, but 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. The first interface is intuitively meant for n different parties (although $\mathcal{Z}$ is universally quantified in our definitions, i.e. it may realise these n parties in an arbitrary fashion); if a message is sent through a channel intended for the i-th party then we write that "party P_i sends / receives a message from Π / $\mathcal{F}$". The closed system $\Pi\|\mathcal{Z}\|\mathcal{A}$ can be executed, let $\textsc{Exec}_{\Pi\|\mathcal{Z}\|\mathcal{A}}$ be the random variable of the output of $\mathcal{Z}$ in such execution.

Definition 1 (UC-secure)**.** A system Π *UC-securely implements* another system Π' if for each $\mathcal{Z}$ and $\mathcal{A}$, there exists an adversary $\mathcal{S}$, such that $\textsc{Exec}_{\Pi\|\mathcal{Z}\|\mathcal{A}}$ and $\textsc{Exec}_{\Pi'\|\mathcal{Z}\|\mathcal{S}}$ are indistinguishable.

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

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

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

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

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

1 : $b := \left[\sigma^e \stackrel{?}{\equiv} \mathsf{Pad}(M) (\mathrm{mod}\ N)\right]$
2 : return b

Figure 1: Multiprime RSA signature scheme

To show that this definition is satisfied, we have to provide the construction of $\mathcal{S}$. Typically, we define a machine *Sim* (the *simulator*) and put $\mathcal{S} := \mathcal{A}\|Sim$. We usually use Def. 1 in settings where Π' is the ideal functionality $\mathcal{F}$ for some cryptographic construction.

An important property of UC is *composability*: if a protocol Π securely implements $\mathcal{F}$, and a different protocol $\Pi'\|\mathcal{F}$ securely implements $\mathcal{F}'$ (we say that "Π' securely implements $\mathcal{F}'$ *in* $\mathcal{F}$*-hybrid model*"), then $\Pi'\|\Pi$ also securely implements $\mathcal{F}'$.

Canetti's framework suggests how the corruption of parties may be modelled [Can20, Sec. 2, 7.1–2]. This modelling is not convenient to use for us, due to the *partial* corruptions that we may have. We thus move the modelling of corruptions from the framework to the definition of individual functionalities and machines, similarly to [PW01].

2.3 Digital Signatures

ECDSAKeyGen$(\mathbb{G}, G, q)$

1 : $x \leftarrow\$ \mathbb{Z}_q$
2 : $Q := x \cdot G$
3 : return $\mathsf{sk} := \langle G, q, x\rangle$, $\mathsf{pk} := \langle Q, G, q\rangle$

ECDSASign(sk, M)

1 : $k \leftarrow\$ \mathbb{Z}_q$
2 : $R := k \cdot G; \quad R := (r_x, r_y)$
3 : $r := r_x \mod q$
4 : $s := k^{-1}(H(M) + r \cdot x) \mod q$
5 : return $\sigma := \langle r, s\rangle$

ECDSAVerify(pk, M, σ)

1 : $v := s^{-1} \mod q$
2 : $u_1 := H(M) \cdot v \mod q$
3 : $u_2 := r \cdot v \mod q$
4 : $R' := u_1 \cdot G + u_2 \cdot Q$
5 : $b := \left[r'_x \mod q \stackrel{?}{=} r\right]$
6 : return b

Figure 2: ECDSA signing

Definition 2 (Multi-prime RSA [Hin08])**.** RSA signature scheme [RSA78] defined by $v \geq 3$ prime numbers is called *multi-prime RSA*. It consists of the three algorithms defined in Figure 1.

In our work, we set parameter as $v = 4$ (currently used in Smart-ID) and refer to such signature variant as 4RSASplit scheme. As the verification algorithm is independent of the number of primes, this RSA variant standard RSA verification algorithm and suitable padding scheme $\mathsf{Pad} : \{0,1\}^* \to \mathbb{Z}_N$ [BR96].

Definition 3 (ECDSA [JMV01])**.** Elliptic Curve Digital Signature Algorithm (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 $\mathbb{F}$. Additionally, a hash function H is fixed as $H : \{0,1\}^* \to \{0, \ldots, 2^{\lceil \log q \rceil} - 1\}$. Subroutines for ECDSA are defined in Figure 2.

Definition 4 (UF-CMA security)**.** A signature scheme is *unforgeable under chosen-message attacks (UF-CMA)* if an adversary with pk and with access to a signing oracle $\mathsf{Sign}(\mathsf{sk}, \cdot)$, where $(\mathsf{sk}, \mathsf{pk})$ are created by the key generation algorithm, has only a negligible chance of producing a message-signature pair (M, σ), such that $\mathsf{Ver}(\mathsf{pk}, M, \sigma)$ accepts, but M was not queried to the signing oracle.

2.4 MtA functionality

Multiplicative-to-Additive (MtA) protocols allow two parties to transform their multiplicative shares into corresponding additive shares. $\mathcal{F}^{\mathsf{MtA}}$ functionality (Figure 3) is a main building block to achieve secure redistribution of an ECDSA ephemeral key k between server and client. Doerner et al. [DKLS18] have shown a protocol that securely realises a variant of $\mathcal{F}^{\mathsf{MtA}}$ in UC using oblivious transfer [KOS15, Roy22]. Theoretically, this functionality could be realised in UC using Paillier encryption [Pai99], Castagnos-Laguillaumie encryption [CL15] and Joye-Libert encryption [BHJL16].

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

Init: On an input (init) from parties P_1 and P_2:

- Send (request-init) to $\mathcal{S}$.
- If respond from $\mathcal{S}$ is (init-success), store (init-complete) and output it to P_1 and P_2. Otherwise, return (init-fail) to P_1 and P_2.

Multiply: Upon receiving $(\mathsf{input}, \mathsf{sid}, x_1 \in \mathbb{Z}_q)$ from P_1 and $(\mathsf{input}, \mathsf{sid}, x_2 \in \mathbb{Z}_q)$ from P_2:

- If init-complete is not stored in a memory and sid is not unique, ignore this query.
- If P_i is corrupted ($i \in \{1, 2\}$), send $(\mathsf{multiply\text{-}share}, x_i, \mathsf{sid})$ to $\mathcal{S}$. Otherwise, send $(\mathsf{multiply\text{-}share}, \mathsf{sid})$ to $\mathcal{S}$.
- Wait for response $(\mathsf{multiply\text{-}share}, \mathsf{sid}, x')$ from $\mathcal{S}$ with $x' \in \{x'_i, \mathsf{fail}\}$. If $x = \mathsf{fail}$, send $\perp$ to P_i and to P_j. Otherwise, set $x_i := x'_i$ and proceed.
- Sample random $y_1 \leftarrow\!\!\$\ \mathbb{Z}_q$.
- Calculate $y_2 := x_1 \cdot x_2 - y_1 \mod q$.
- If P_i is corrupted ($i \in \{1, 2\}$), send $(\mathsf{multiply\text{-}share\text{-}output}, y_i, \mathsf{sid})$ to $\mathcal{S}$.
- Wait for response $(\mathsf{multiply\text{-}share\text{-}output}, \mathsf{sid}, y'_i)$ from $\mathcal{S}$. Set $y_i := y'_i$.
- Send $(\mathsf{output}, \mathsf{sid}, y_1)$ to P_1 and $(\mathsf{output}, \mathsf{sid}, y_2)$ to P_2.

Figure 3: $\mathcal{F}^{\mathsf{MtA}}$: ideal functionality for transforming multiplicative shares into additive.

2.5 Non-interactive Zero-Knowledge Proofs

Non-interactive zero-knowledge proofs (NIZKPs) are the family of protocols that allows a prover to demonstrate to a verifier that the former knows some value w, for which a relation $R(x, w)$ holds, without requiring interactive communication between the parties. NIZKP of knowledge (NIZKPoK) of discrete logarithm is a common tool to construct distributed ECDSA [DKLS18, CCL+19, XAX+21, DKLS19, CGG+20, Lin21] protocols. One could instantiate NIZKP from an interactive Σ-protocol using transforms such as Fiat-Shamir [FS87] (FS), Pass [Pas03], Fischlinn [Fis05] and Chen et al [CDG+25]. The most common approach to construct NIZKPoK of discrete logarithm is applying FS transform to Schnorr identification scheme [Sch90]. In this case, however, knowledge extraction algorithm $\mathsf{Extract}(\cdot)$ cannot be straightforwardly instantiated for FS-based proofs in the UC due to the requirement of simulator rewinding. Lysyanskaya and Rosenbloom [LR22] have shown how to instantiate NIZKP in the UC model using Fischlin transform to obtain extractable and simulatable NIZKPs. Chen et al.[CDG+25] demonstrated that NIZKPoK can be instantiated with their transform in UC model as well. Corresponding ideal functionality $\mathcal{F}^{\mathsf{NIZKP}}$ is given in Figure 2.5.

2.6 Random Oracle Model

Random Oracle Model (ROM) could be used to simulate the behavior of hash functions in signing protocols [BR93]. In the ROM, hash functions are abstracted to an external oracle that behaves like a truly random function, providing consistent and unpredictable responses to the parties with access to it. To represent the access of random oracle to several parties, we use *Restricted Programmable Observable Global Random Oracle* model introduced by Camenisch et al [CDG+18]. The corresponding ideal functionality $\mathcal{F}^{\mathsf{rpoRO}}$ is presented in Figure 2.6.

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

Setup: Upon receiving $(\mathsf{setup}, \mathsf{sid})$ from a party P:

- If sid does not correspond to a session, ignore this query.
- Send $(\mathsf{setup}, \mathsf{sid})$ to adversary $\mathcal{S}$.
- Upon receiving from adversary $\mathcal{S}$ the tuple $(\mathsf{algorithms}, \mathsf{sid}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim\text{-}Setup}, \mathsf{Sim\text{-}Prove}, \mathsf{Extract})$ with algorithm descriptions, store this tuple.

Prove: Upon receiving $(\mathsf{prove}, \mathsf{sid}, x, w)$ from a party P:

- If $R(x, w) \neq 1$ or sid does not correspond to a session, ignore this query.
- Compute π using $\mathsf{Sim\text{-}Setup}$ and $\mathsf{Sim\text{-}Prove}$ algorithms.
- Run $\mathsf{Verify}(x, \pi)$. If it returns 1, record and output the message $(\mathsf{proof}, \mathsf{sid}, x, \pi)$. Otherwise, output (fail).

Verify: Upon receiving $(\mathsf{verify}, \mathsf{sid}, x, \pi)$ from a party P:

- If sid does not correspond to a session, ignore this query.
- If $\mathsf{Verify}(x, \pi) = 0$, output $(\mathsf{verification}, \mathsf{sid}, x, \pi, 0)$.
- Otherwise if $(\mathsf{proof}, \mathsf{sid}, x, \pi)$ is already stored, output $(\mathsf{verification}, \mathsf{sid}, x, \pi, 1)$.
- Otherwise, run $\mathsf{Extract}$ to compute w. If $R(x, w) = 1$, output $(\mathsf{verification}, \mathsf{sid}, x, \pi, 1)$. If $R(x, w) = 0$, output (fail).

Figure 4: $\mathcal{F}^{\mathsf{NIZKP}}$: ideal functionality for non-interactive zero-knowledge proof of knowledge.

3 Ideal Functionalities for Server-Supported Signing

In this section, we define two ideal functionalities for server-supported signing: $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. The first functionality $\mathcal{F}^{\mathsf{gSpl}}$ is the generic functionality for server-supported signing, which is instantiated for n parties, where the first two parties (called client and server) can run key generation/signing and all parties can run signature verification. The second functionality $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ is instantiated for two parties (called client and server) and takes as a parameter any signature scheme Sig.

The two functionalities are best understood as addressing distinct modeling objectives. The generic ideal functionality $\mathcal{F}^{\mathsf{gSpl}}$ abstracts signing in a scheme-independent way: it captures the fact that signing is performed in a server-supported manner, without requiring the resulting public keys and signatures to match the syntax or verification procedure of any particular signature scheme. This abstraction is appropriate when the signatures are embedded in a part of some composable system and interoperability with an outside verifier is irrelevant. By contrast, definition of $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ requires generated keys and signatures conforming to a concrete signature scheme Sig (i.e. functionality output is verifiable by $\mathsf{Sig.Ver}(\cdot)$). This stronger functionality is the appropriate abstraction when server-supported signing must remain compatible with standard signature scheme.

We show the relation between two functionalities by proving that there exists a trivial protocol securely realizing $\mathcal{F}^{\mathsf{gSpl}}$ in $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$-hybrid model for any UF-CMA secure signature scheme Sig (Section A).

3.1 Ideal Functionality $\mathcal{F}^{\mathsf{gSpl}}$

$\mathcal{F}^{\mathsf{gSpl}}$ is the ideal functionality for server-supported signatures (Figure 8). It handles the PIN-s (i.e. low-entropy passwords) that are used to authenticate the client to the server. It also handles clone detection (Figures 6 and 7). It is partly inspired by the

$\mathcal{F}^{\mathsf{rpoRO}}$: 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.

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

Observe: Upon receiving input $(\mathsf{observe}, \mathsf{sid})$ from the adversary $\mathcal{S}$:

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

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

- 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 $(\mathsf{program\text{-}confirmed})$.

IsProgrammed: Upon receiving $(\mathsf{is\text{-}programmed}, \mathsf{sid}, m)$ from a party P:

- If sid does not correspond to current session, ignore this query.
- If $m \in \mathsf{prog}$, set $b := 1$, else $b := 0$.
- Output $(\mathsf{is\text{-}programmed}, b)$

Figure 5: $\mathcal{F}^{\mathsf{rpoRO}}$, ideal functionality for Restricted Programmable Observable Global Random Oracle (rpGRO)

ideal threshold signature functionality of Canetti et al. [CGG+20]. 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 are identified as *the client* and *the server*, capable of cooperatively producing signatures, while any party can verify the signatures.

The internal state of $\mathcal{F}^{\mathsf{gSpl}}$ contains a set of tuples (M, σ, b), recording all the signatures that $\mathcal{F}^{\mathsf{gSpl}}$ has created or verified. The tuple $(M, \sigma, 1)$ in the state of $\mathcal{F}^{\mathsf{gSpl}}$ means that σ is considered a valid signature for M, while $(M, \sigma, 0)$ denotes σ being an invalid signature for M. This part of $\mathcal{F}^{\mathsf{gSpl}}$ is identical to the other ideal functionalities modeling signatures.

Corruption level: The functionality $\mathcal{F}^{\mathsf{gSpl}}$ tracks the corruption levels of parties, and the source of the last signing query (necessary for modeling clone detection). The corruption level c_S of the 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_C of the client is more fine-grained: here "0" stands for an uncorrupted client and "3" means that the adversary has full control over it. There are also intermediate levels: $\mathsf{c}_\mathrm{C} = 1$ means that the adversary has managed to learn client's memory sometime between signing sessions, and $\mathsf{c}_\mathrm{C} = 2$ means that client's memory has leaked during a signing session. In the former case, the memory has been "encrypted" with a PIN. In the latter case, the adversary has learned the key material that the client uses for signing. In cases "1" and "2", the adversary does not have control over the client's device that would allow it to control the messages the client sends out.

Our treatment of corruptions departs from the commonly used binary static/adaptive formulations in the UC framework, though, only in granularity. Binary *honest/corrupt* abstraction would be too coarse for desired threat model because the relevant compromise events on the phone have different consequences. In particular, the treatment of low-entropy PIN protection and clone detection requires distinguishing leakage of encrypted memory

Initialization of $\mathcal{F}^{\mathsf{gSpl}}$: Set $\mathsf{c_S} := \mathsf{c_C} := T := 0$; $b_{\mathrm{lq}} := b_{\mathrm{OK}} := 1$; $\mathsf{Ver} := \bot$.
Initialization of $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$: Set $\mathsf{c_S} := \mathsf{c_C} := T := 0$; $b_{\mathrm{lq}} := b_{\mathrm{OK}} := 1$, $\mathsf{pk} = \mathsf{sk} = \bot$.

Derived values for $\mathcal{F}^{\mathsf{gSpl}}$: $b_{\mathrm{KG}} \equiv [\mathsf{Ver} \neq \bot]$. $b_{\mathrm{keyleak}} \equiv [\mathsf{c_S} = 1] \wedge [\mathsf{c_C} \geq 1]$.
Derived values for $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$: $b_{\mathrm{KG}} \equiv [\mathsf{pk} \neq \bot]$. $b_{\mathrm{keyleak}} \equiv [\mathsf{c_S} = 1] \wedge [\mathsf{c_C} \geq 1]$.

Corrupt server: On input (corrupt-server) from $\mathcal{S}$: set $\mathsf{c_S} := 1$. Send (corrupt-server) to server.
Corrupt client: On input (corrupt-client, ℓ) from $\mathcal{S}$, where $\ell \in \{1, 2, 3\}$:

- If $\mathsf{c_C} > \ell$, ignore this query.
- Set $b_{\mathrm{lq}} := \bot$. If $b_{\mathrm{KG}} = 0$, set $\mathsf{c_C} := 3$. Otherwise, set $\mathsf{c_C} := \ell$. Send (corrupt-client, c_{C}) to client.

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

Figure 6: Initialization and corruption instructions for $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ functionalities

from leakage of unencrypted signing state, and both from full corruption of the device. For this reason, we encode these cases as distinct corruption levels of the phone, rather than subsuming them under a single corrupted state.

Tracking origin of queries: The internal state of functionality contains the bit b_{lq} for indicating whether the client ($b_{\mathrm{lq}} = 1$) or the adversary ($b_{\mathrm{lq}} = 0$) made the last signing query. Whenever the client is corrupted, this bit is set to $\bot$, allowing the next signing query to come from either of them. Generally, corrupting the client means ability to read its memory, and this presumably includes the current value of the bitstring used for clone detection.

Simulating PIN: The internal state of $\mathcal{F}^{\mathsf{gSpl}}$ 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 client corruption level $\mathsf{c_C} = 1$, therefore it has to be modeled 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 $\mathsf{c_C} := 2$.

process-pin(PIN′):

- If PIN = PIN′, set $T := 0$ and return execution to the caller. Otherwise, increment T.
- If $T \geq T_0$ then set $b_{\mathrm{OK}} := 0$.
- Return (sign-fail) to the invoker(s).

clone-check(d):

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

request-sig(sid, M):

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

Figure 7: Supporting routines/algorithms for ideal functionalities $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$

Clone detection and offline-guessing attacks protection: The internal state of $\mathcal{F}^{\mathsf{gSpl}}$ contains the bit b_{OK}. The server having detected several clones of the client is indicated by setting $b_{\mathrm{OK}} := 0$. When this happens, the server no longer participates in signature creation.

Leaking the private key. The derived bit b_{keyleak} indicates whether the entire signing key may have leaked to the adversary. This leak happens when the adversary has sufficiently corrupted both the client and the server, i.e. when $c_{\mathrm{C}} = 2$ (the adversary knows client's keyshare) and $c_{\mathrm{S}} = 1$ (the adversary knows server's keyshare). Note also that if the adversary has server's keyshare and client's encrypted memory, then it can try out all possible PIN's to decrypt the latter; it can verify it's guess of the PIN with the help of the former. In this process, the adversary also learns the correct PIN. Hence the setting $c_{\mathrm{C}} = c_{\mathrm{S}} = 1$ is not possible; we immediately move to $c_{\mathrm{C}} = 2$.

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

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

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

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

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

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

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

1. Call $\mathsf{clone\text{-}check}(0)$ and $\mathsf{process\text{-}pin}(\mathsf{PIN}')$. If successful, set $\mathsf{c_C} := 2$ and go to step 3.
2. Call $\mathsf{clone\text{-}check}(0)$. If successful, go to step 3.
3. Compute σ using $\mathsf{request\text{-}sig}(\mathsf{sid}, M)$. If $\sigma \neq \perp$, output $(\mathsf{sign\text{-}success}, \mathsf{sid}, M, \sigma)$ to server. Otherwise, output $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to both $\mathcal{S}$ and server.

Verification: On input $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ from a party P, ignore, unless sid is unique. Otherwise:

- Define the bit b as follows:
  - If (M, σ, b') is recorded, then set $b := b'$.
  - If $(M, \sigma', 1)$ record does not exist for any σ' and $b_{\mathrm{keyleak}} = 0$, then set $b := 0$.
  - Else, set $b := \mathsf{Ver}(M, \sigma)$.
- Record (M, σ, b) and output $(\mathsf{is\text{-}verified}, \mathsf{sid}, M, \sigma, b)$ to party P.

Figure 8: Ideal functionality for server-supported signing $\mathcal{F}^{\mathsf{gSpl}}$

In order to support the validity of the construction of $\mathcal{F}^{\mathsf{gSpl}}$, we show that it satisfies a number of properties analogous to those presented by Buldas et al. [BKLO17], and that its improved clone detection system works. These properties state something about the

possible input-output behaviours of $\mathcal{F}^{\mathsf{gSpl}}$; these restrictions of the behaviour correspond to our understanding of "security".

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

Proof. Let us assume an adversary $\mathcal{S}$ produced a pair (M, σ), such that a query $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ by some party P returns $(\mathsf{is\text{-}verified}, sid, M, \sigma, 1)$. According to the definition $\mathcal{F}^{\mathsf{gSpl}}$, 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 $\mathcal{F}^{\mathsf{gSpl}}$.
2. If $b_{\text{keyleak}} = 1$ and adversary recorded a pair (M, σ) such that $\mathsf{Ver}(\sigma, M) = 1$.

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

Theorem 2. *An adversary $\mathcal{S}$ and an environment $\mathcal{Z}$, running in parallel with $\mathcal{M}_{\text{PP}} \| \mathcal{F}^{gSpl}$, not corrupting the server, and corrupting the client 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 $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ by some party would return $(\mathsf{is\text{-}verified}, \mathsf{sid}, M, \sigma, 1)$, but M was never submitted to $\mathcal{F}^{gSpl}$ by the client.*

Our second theorem characterizes the security of the mechanisms limiting adversary's guesses of the PIN. Note that these mechanisms involve both the counter T in Figures 6-7, 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 $\mathcal{M}_{\text{PP}}$ whose task is to permute the possible PINs that the client submits to $\mathcal{F}^{\mathsf{gSpl}}$. In this way, the "*real*" PIN is not known by $\mathcal{Z}$.

The machine $\mathcal{M}_{\text{PP}}$ has the interface that allows it to be placed between the client (as part of $\mathcal{Z}$) and the functionality $\mathcal{F}^{\mathsf{gSpl}}$ (or any protocol implementing $\mathcal{F}^{\mathsf{gSpl}}$). The machine $\mathcal{M}_{\text{PP}}$ is defined in Figure 9.

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

Initialise: On input $(\mathsf{init}, L, T_0, \mathsf{PIN})$ from the client:

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

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

Figure 9: Description of supporting PIN interception machine $\mathcal{M}_{\text{PP}}$

Proof. Let us assume an adversary $\mathcal{S}$ produced a pair (M, σ), such that a query $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ by some party returns $(\mathsf{verified}, \mathsf{sid}, M, \sigma, 1)$. According to the definition $\mathcal{F}^{\mathsf{gSpl}}$, 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 $\mathcal{F}^{\mathsf{gSpl}}$. In this case, $\mathcal{S}$ had to guess PIN. $\mathcal{S}$ can make PIN guessing attempts, when $\mathsf{c_C} = 1$. With each command, $\mathcal{S}$ can test only one value of

PIN. When $\mathcal{S}$ submits wrong guess, $\mathcal{F}^{\mathsf{gSpl}}$ increments counter T until it reaches T_0 and $\mathcal{F}^{\mathsf{gSpl}}$ stops. The mechanisms for reducing T without guessing the PIN are the following:

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

2. If $b_{\mathrm{keyleak}} = 1$ and adversary submitted a pair (M, σ) such that $\mathsf{Ver}(\sigma, M) = 1$ during verification command.

First case implies that M was submitted to $\mathcal{F}^{\mathsf{gSpl}}$ by either client or $\mathcal{S}$, where $\mathcal{S}$ could have submitted M to $\mathcal{F}^{\mathsf{gSpl}}$ with probability at most T_0/L. 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 $\mathcal{S}$ and an environment $\mathcal{Z}$ are running in parallel with $\mathcal{F}^{\mathsf{gSpl}}$, not corrupting the server and corrupting the client to at most level 2. Suppose that $\mathcal{S}$ and $\mathcal{Z}$ cause $\mathcal{F}^{\mathsf{gSpl}}$ to be initialized, after which (not necessarily immediately) they issue commands for Signing by client and server, and for Signing by client 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 client to level 3. Then, it is impossible for $\mathcal{S}$ and $\mathcal{Z}$ to produce a pair (M, σ), such that a query $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ by some party would return $(\mathsf{is\text{-}verified}, \mathsf{sid}, M, \sigma, 1)$, but M was not submitted for signing to $\mathcal{F}^{\mathsf{gSpl}}$ 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. [BKLO17] specified for their protocol that the random string chosen for clone detection is generated by the client. Sarr [Sar19] showed an attack which gives an adversary multiple attempts at guessing the PIN, by resetting the string value after $(L-1)$ tries to the old value. They proposed an improvement, changing the protocol to make the server generate the clone detection string. However, Sarr [Sar19] did not give a security definition that such construction could aim to satisfy.

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

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

2. If $b_{\mathrm{keyleak}} = 1$ and adversary recorded a pair (M, σ) such that $\mathsf{Ver}(\sigma, M) = 1$ during the verification command.

First case implies that M was submitted to $\mathcal{F}^{\mathsf{gSpl}}$ by either client or $\mathcal{S}$. This suggests that $\mathcal{S}$ could have submitted M to $\mathcal{F}^{\mathsf{gSpl}}$ only if client has not issued signing commands after $\mathcal{S}$. It would mean that clone detection mechanism is activated and $\mathcal{S}$ cannot issue signing commands anymore. Second case implies that both parties are corrupted by adversary. Both of these cases contradict with the given assumptions. □

3.2 Ideal Functionality $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$

Next, we define functionality $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ (Figure 10) which is less general and more similar to the ideal functionalities defined for threshold/multisignature protocols defined in the literature [CGG+20, DKLS24, CEN25, DKLS25]. It is parametrized by a signature scheme Sig, and models the distributed key and signature generations of that scheme. There are lot of similarities between $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$, including the corruption levels of the client and the server, the PIN verification and the clone detection mechanisms. Noteworthy differences are in the generation of keys and signatures, where $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ uses the methods of signature scheme Sig, instead of invoking the adversary $\mathcal{S}$.

Key Generation: On input $(\mathsf{keygen}, \mathsf{sid}, L, T_0, \mathsf{PIN})$ from the client and command $(\mathsf{keygen}, \mathsf{sid}, T_0)$ from the server, if $(\mathsf{pk}, \mathsf{sk})$ are already defined, sid is not unique, the values T_0 differ or $\mathsf{PIN} \notin \{1, \ldots, L\}$, then ignore this query. Otherwise:

- Generate $(\mathsf{pk}, \mathsf{sk})$ by running $\mathsf{KGen}()$ and send $(\mathsf{keygen\text{-}start}, \mathsf{pk})$ to $\mathcal{S}$.
- If $\mathcal{S}$ returns $(\mathsf{keygen\text{-}stop})$, thensend $\perp$ to both client and server.
- If $\mathcal{S}$ returns $(\mathsf{keygen\text{-}ok})$, then store L, PIN, and T_0, and send pk to both client and server.

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

- Call $\mathsf{process\text{-}pin}(\mathsf{PIN}')$ and $\mathsf{clone\text{-}check}(1)$. If successful, compute σ by running $\mathsf{Sign}(\mathsf{sk}, M)$,
- Send $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ to $\mathcal{S}$. On input $(\mathsf{sign\text{-}proceed}, \mathsf{sid})$ from $\mathcal{S}$, send $(\mathsf{sign\text{-}success}, \mathsf{sid}, M, \sigma)$ to client and $(\mathsf{signature}, \mathsf{sid}, M, \sigma)$ to server.

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

- Call $\mathsf{process\text{-}pin}(\mathsf{PIN}')$ and then compute σ by running $\mathsf{Sign}(\mathsf{sk}, M)$.
- Send $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ to $\mathcal{S}$. On input $(\mathsf{sign\text{-}proceed}, \mathsf{sid})$ from $\mathcal{S}$, send $(\mathsf{sign\text{-}success}, \mathsf{sid}, M, \sigma)$ to client.

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

1. Call $\mathsf{process\text{-}pin}(\mathsf{PIN}')$. Upon success, set $T := 0$, $\mathsf{c}_{\mathrm{C}} := 2$ and go to step 2. Otherwise, call $\mathsf{clone\text{-}check}(0)$ and return $(\mathsf{sign\text{-}fail})$ to $\mathcal{S}$ and server.
2. Call $\mathsf{clone\text{-}check}(0)$. If successful, go to step 3.
3. Compute σ by running $\mathsf{Sign}(\mathsf{sk}, M)$. Send $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ to $\mathcal{S}$. On input $(\mathsf{sign\text{-}proceed}, \mathsf{sid})$ from $\mathcal{S}$, send $(\mathsf{signature}, \mathsf{sid}, M, \sigma)$ to server.

Figure 10: $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$, ideal functionality for server-supported signing with scheme Sig

The analogues of Theorems 1–3 can be stated for $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$. With theorems 4–6, we show that our ideal functionality $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ has properties similar to $\mathcal{F}^{\mathsf{gSpl}}$, correctly capturing the security requirements that we set for server-supported signature schemes. The proofs of the following theorems have significant differences from the proofs of the previous ones, because while the properties of $\mathcal{F}^{\mathsf{gSpl}}$ where guaranteed by the way it added the tuples (M, σ, b) to its internal state, the properties of $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ largely rely on the security of Sig.

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

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

- Upon initialization, $\mathcal{B}$ does not make $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ 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 $\mathsf{Sign}(\mathsf{sk}, M)$, the machine $\mathcal{B}$ submits M to its oracle and receives back σ.

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

Theorem 5. *Suppose that an adversary $\mathcal{S}$ and an environment $\mathcal{Z}$ running in parallel with $\mathcal{M}_{\mathrm{PP}} \| \mathcal{F}_{Sig}^{Spl}$, not corrupting the server and corrupting the client to at most level 1, have the probability p of producing a pair (M, σ) such that $\mathsf{Ver}(\mathsf{pk}, M, \sigma)$ accepts, but M was never submitted to $\mathcal{F}_{Sig}^{Spl}$ by the client. If $(p - T_0/L)$ is non-negligible (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 $\mathcal{B}^{\mathsf{Sign}(\mathsf{sk},\cdot)}(\mathsf{pk})$ that will break the UF-CMA security of Sig. The machine $\mathcal{B}$ executes $\mathcal{M}_{\mathrm{PP}} || \mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}} || \mathcal{Z} || \mathcal{S}$ internally, except:

- Upon initialisation, instead of generating keypair as defined in $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$, use key pk that was supplied as input to $\mathcal{B}$. No sk gets generated by $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$. Instead of receiving client-supplied PIN, $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ receives and stores $\pi(\mathsf{PIN})$ from $\mathcal{M}_{\mathrm{PP}}$.
- Upon receiving signing command, instead of computing signature σ via $\mathsf{Sign}(\mathsf{sk}, M)$, 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 $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ by the client. In order to produce such pair via submitting M to $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ by $\mathcal{S}$, it must guess the PIN. The guessing is enabled if $\mathsf{c}_{\mathrm{C}} = 1$ and at each guess, the adversary can test one possible value of the PIN. At each wrong guess, $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ 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_{C}, after which PIN is revealed to $\mathcal{S}$ and wrong guesses of PIN are no longer tracked. We have presumed that this does not happen.
- Let the client 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 $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ 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 $\mathcal{S}$ and an environment $\mathcal{Z}$ are running in parallel with $\mathcal{F}_{Sig}^{Spl}$, not corrupting the server and corrupting the client to at most level 2. Suppose that $\mathcal{S}$ and $\mathcal{Z}$ cause $\mathcal{F}_{Sig}^{Spl}$ to be initialized (returning public key pk), after which (not*

necessarily immediately) they issue commands for Signing by client and server, and for Signing by client 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 client. If $\mathcal{S}$ and $\mathcal{Z}$ have non-negligible probability of producing a pair (M, σ), such that $\mathsf{Ver}(\mathsf{pk}, M, \sigma)$ accepts, but M was not submitted for signing to $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ before the second of the above command sets was issued, then Sig is not UF-CMA secure.

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

- Upon initialisation, instead of generating keypair as defined in $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$, use key pk that was supplied as input to $\mathcal{B}$. No sk gets generated by $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$. Instead of receiving client-supplied PIN, $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ receives ans stores $\pi(\mathsf{PIN})$ from $\mathcal{M}_{\mathrm{PP}}$.
- Upon receiving signing command, instead of computing signature as $\sigma = \mathsf{Sign}(\mathsf{sk}, M)$, make query tho 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 $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ it had to be submitted to $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ by either the client or by $\mathcal{S}$. By our assumption, there was non-negligible probability that M was either not submitted to $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ at all, or it was submitted after the command sets for Signing by client 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 client). In the former case, $\mathcal{B}$ has broken the UF-CMA security of Sig. In the latter case, clone detection mechanism was activated in $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ and b_{OK} was set to 0 which means that the adversary was unable to issue signing commands anymore.

□

4 Server-Supported RSA Signatures

In this section, we describe a server-supported RSA signing protocol $\Pi^{\mathsf{4RSASplit}}$ proposed by Buldas et al. [BKLO17] with improved clone detection algorithm addressing Sarr's critique [Sar19].

The server-supported RSA protocol is run between two parties: client C and server S in the presence of adversary $\mathcal{A}$. Buldas et al. [BKLO17] indicate that since client's side of the protocol is run on the constrained device, the cryptographic material on the client's side may not be protected as strongly as on the server's side. Considering adversary may infect user's device with malware, there exists a possibility of client's device memory leakage. If it occurs while the key generation or signing process are running, adversary could learn unencrypted memory obtaining a client's private key share. However, if this attack is executed when no aforementioned protocols are running, adversary gets encrypted memory, that corresponds to learning seed w. Additionally, to deter offline-guessing attack, server counts each incorrect PIN entry by client and aborts any further communication if number of attempts exceeds defined threshold. Finally, to counter impersonation attacks from memory leaks, the clone detection mechanism is employed to identify and prevent unauthorized replication of client's device memory.

4.1 Key Generation

Before running key generation protocol (Figure 12), set of parameters such as security parameter λ, maximum amount of sequential failed PIN queries T_0 and RSA public

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

1 : for $i \in \{0, 1, \ldots, 255\}$ do
2 : $\quad x \coloneqq F_{\mathsf{PIN}}(u \,||\, i)$
3 : $\quad$ if $x < \mathsf{size}$,
4 : $\quad\quad$ return x
5 : return FAIL

Figure 11: Supporting algorithm for a key splitting $\mathsf{genShare}(u, \mathsf{PIN}, \mathsf{size})$

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

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

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

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

Figure 12: Server-supported RSA key generation

exponent e are predefined. For key generation, client 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. Both parties locally compute their private exponents as $d_i = e^{-1} \pmod{\phi(n_i)}$. C also generates a random string u, takes PIN proved by user and uses it to compute a value $d_1' = \mathsf{genShare}(u, \mathsf{PIN}, n_1) \in \mathbb{Z}_{n_1}$ (Figure. S generates a bitstring w for clone detection purposes. Then client C sends n_1 and $d_1'' = (d_1 - d_1') \bmod \phi(n_1)$ to server S; the latter sends back w and n. C verifies whether $n_1 \mid n$ and upon success stores $n_1, u, n, e, w' := w$, deleting d_1, d_1', d_1'' and the prime factors of n_1. S stores $n, n_1, n_2, d_1'', d_2, w$ and initializes the incorrect PIN counter as $T \coloneqq 0$. The resulting public key is $\mathsf{pk} = (n_1 \cdot n_2, e)$.

The value d_1' is derived using genShare function (Figure 11). It expands user-provided PIN into a longer value, using the randomness u that will be leaked if $\mathcal{A}$ obtains client's encrypted memory. The algorithm calls a pseudorandom function that could be instantiated from a block cipher or extendable-output function.

4.2 Signing

In order to sign a message M using a user-provided PIN, client C computes $d_1' \leftarrow \mathsf{genShare}(u, \mathsf{PIN}, n_1)$, $y \coloneqq \mathsf{Pad}(M)^{d_1'} \bmod n_1$, and sends (y, w') to S. Server S checks that received w' is equal to stored bitstring w and whether the client did not exceed allowed number of attempts T_0, halting if it is not the case. S computes $\sigma_1 \leftarrow y \cdot \mathsf{Pad}(M)^{d_1''} \bmod n_1$ and verifies it against client's modulus n_1. If σ is not a valid signature, then S increments T and restarts the protocol. Otherwise, they compute σ using the Chinese Remainder Theorem from values σ_1 and $\mathsf{Pad}(M)^{d_2} \bmod n_2$. Finally, S generates a bitstring w', S sets

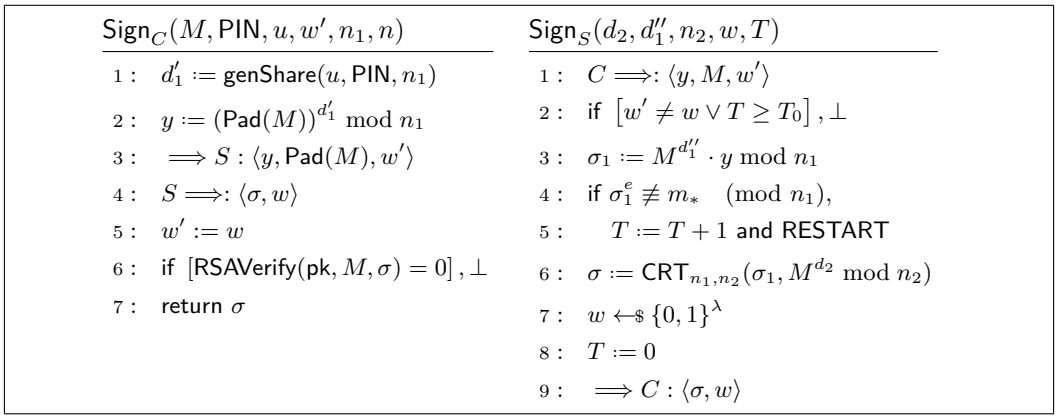

Figure 13: Server-Supported RSA Signing

$T := 0$, and sends (σ, w') back to the client C. On their side C updates clone detection bitstring $w' := w$ and outputs a signature σ.

5 Security of Server-Supported RSA

We prove that $\Pi^{\mathsf{4RSASplit}}$ protocol securely realises $\mathcal{F}^{\mathsf{gSpl}}$ in UC model. We have to analyse several adversarial scenarios and the corresponding outcomes. If adversary $\mathcal{A}$ corrupts server, it obtains values d_2, d_2'', w, n_1, n_2. If adversary $\mathcal{A}$ get access to client's encrypted memory, it possesses values u, w, n_1. If adversary $\mathcal{A}$ get access to client's unencrypted memory, it get a hold of values u, d_1', w, n_1.

To address various adversarial scenarios properly, we model client'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.

Theorem 7. *If RSA with padding function* Pad *is UF-CMA secure, then server-supported RSA signing presented in Figures 12 and 13 is a secure implementation of* $\mathcal{F}^{\mathsf{gSpl}}$.

In order to show that $\Pi^{\mathsf{4RSASplit}}$ is at least as secure as $\mathcal{F}^{\mathsf{gSpl}}$, we will construct a simulator *Sim*, such that for any adversary $\mathcal{A}$, no environment $\mathcal{Z}$ can distinguish an execution with $\Pi^{\mathsf{4RSASplit}} \| \mathcal{A}$ from an execution with $\mathcal{F}^{\mathsf{gSpl}} \| Sim \| \mathcal{A}$. The state of the simulator contains the values mentioned in the description of the ideal functionality and the protocols, namely:

$$\mathsf{c_S}, \mathsf{c_C}, e, n, n_1, n_2, d_1, d_1', d_1'', d_2, u, T, \mathsf{PIN}, w \ .$$

Most of these values are undefined at the start of an execution. Actually, only $\mathsf{c_C} := 0$ and $\mathsf{c_S} := 0$ are defined at the beginning, and the value e is defined during the initialization. Other values become defined while simulating key generation and also when the corruption level of the client or the server increases. Tables 1–3 give an overview of when each of these other values is initialized. In these tables, the values in the left / bottom cell are initialized during key generation. The rest are initialized when the corruption level of the client or the server increases. Recall that the case $\mathsf{c_C} = \mathsf{c_S} = 1$ is impossible.

Below, we outline the manner in which the simulator interacts with and reacts to commands from $\mathcal{F}^{\mathsf{gSpl}}$.

5.1 Initialization

During the initialization, simulator defines the public exponent e and sets $T = 0$.

Table 1: The values that are defined in the simulator, if both the server and the client were uncorrupted during key generation. For each $(\mathsf{c_C}, \mathsf{c_S})$ value, the cells to the left and below are accounted for, too.

	$\mathsf{c_S} = 0$	$\mathsf{c_S} = 1$
$\mathsf{c_C} \geq 2$	PIN, d_1', d_1''	(empty)
$\mathsf{c_C} = 1$	u,	N/A. Go to $\mathsf{c_C} \geq 2$
$\mathsf{c_C} = 0$	n, n_1, $\phi(n_1)$, n_2, d_1, d_2, T, w	d_1', d_1''

Table 2: The values that are defined in the simulator, if the server was corrupted during key generation

$\mathsf{c_C} \geq 2$	PIN, u
$\mathsf{c_C} = 1$	N/A. Go to $\mathsf{c_C} \geq 2$
$\mathsf{c_C} = 0$	n, n_1, $\phi(n_1)$, n_2, d_1, d_1', d_1'', T, w

5.2 Corruption before key generation

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

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

5.3 Key generation

On command (keygen-init) from $\mathcal{F}^{\mathsf{gSpl}}$, the simulator proceeds as follows; depending on the corruption level of the client and server.

5.3.1 Neither is corrupted.

Simulate the execution of both client and server inside *Sim*, running the distributed key generation protocol in Fig. 12. During the execution, initialize only the values named in the cell $\mathsf{c_P} = \mathsf{c_S} = 0$ in Table 1; the outputs of the key generation protocol can be computed from these values. In this case, the real protocol execution is identical to the simulated one, since all values are generated in the same way as in the real protocol. Send $(\mathsf{key}, \mathsf{Ver})$ back to $\mathcal{F}^{\mathsf{gSpl}}$, where Ver is the description of the $\mathsf{RSAVerify}(\cdot)$ algorithm using the public key just generated.

5.3.2 Client is corrupted.

Simulate the server's execution of the protocol, defining the values in the "$\mathsf{c_S} = 0$" cell of Table 3. Generate RSA modulus n_2, compute private exponent $d_2 := e^{-1} \mod \phi(n_2)$. Upon receiving $\langle d_1'', n_1 \rangle$ from $\mathcal{A}$ (as if coming from the corrupted client), compute $n := n_1 \cdot n_2$ and sample clone detection bitstring $w := \{0,1\}^*$. Send $\langle n, w \rangle$ to the corrupted client, i.e. to $\mathcal{A}$.

Let Ver be the description of the $\mathsf{RSAVerify}(\cdot)$ verification algorithm using the public key just generated. Send $(\mathsf{key}, \mathsf{Ver})$ back to $\mathcal{F}^{\mathsf{gSpl}}$.

Table 3: The values that are defined in the simulator, if the client was corrupted during key generation

$\mathsf{c_S} = 0$	$\mathsf{c_S} = 1$
n, n_1, n_2, d_2, d_1'', T, PIN, w	(empty)

The key generation with corrupted client triggers the leakage of client's PIN. Hence receive $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{gSpl}}$.

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

5.3.3 Server is corrupted.

Simulate client's execution of the protocol, defining the values in the "$\mathsf{c}_\mathrm{C} = 0$" cell of Table 2. Generate RSA modulus n_1, compute private $d_1 := e^{-1} \mod \phi(n_1)$ and sample randomly $d_1'' := \mathbb{Z}_{\phi(n_1)}$. Send $\langle d_1'', n_1 \rangle$ to the corrupted server, i.e. to $\mathcal{A}$. Upon receiving $\langle n, w \rangle$ from $\mathcal{A}$, check whether $n_1 \mid n$.

Let Ver be the description of the $\mathsf{RSAVerify}(\cdot)$ algorithm using the public key just generated. Send $(\mathsf{key}, \mathsf{Ver})$ back to $\mathcal{F}^{\mathsf{gSpl}}$.

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

5.3.4 Both parties are corrupted.

In this case, the adversary $\mathcal{A}$ runs the key generation protocol. At the end of the protocol, the adversary tells *Sim* what the machines $\mathcal{M}_{\mathrm{cli}}$ and $\mathcal{M}_{\mathrm{srv}}$ should output to the environment. The simulator extracts the public key pk from this command, defines Ver as the description of the $\mathsf{RSAVerify}(\cdot)$ verification algorithm using pk, and sends $(\mathsf{key}, \mathsf{Ver})$ back to $\mathcal{F}^{\mathsf{gSpl}}$.

5.4 Corruption after key generation

5.4.1 Corrupting the server.

On command "Corrupt server" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{srv}}$, after the keys have been generated, send $(\mathsf{corrupt\text{-}server})$ to $\mathcal{F}^{\mathsf{gSpl}}$ and proceed as follows:

- If $\mathsf{c}_\mathsf{C} = 0$, then we move from cell $(\mathsf{c}_\mathrm{C}, \mathsf{c}_\mathrm{S}) = (0, 0)$ to cell $(0, 1)$ in Table 1. Generate random d_1', d_1'', subject to $(d_1' + d_1'') \bmod \phi(n_1) = d_1$. Send the state of the server to $\mathcal{A}$.

- If $\mathsf{c}_\mathsf{C} = 1$, then we move from cell $(\mathsf{c}_\mathrm{C}, \mathsf{c}_\mathrm{S}) = (1, 0)$ to cell $(2, 1)$ in Table 1. I.e., $\mathcal{F}^{\mathsf{gSpl}}$ sets $\mathsf{c}_\mathsf{C} := 2$ and the simulator receives $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{gSpl}}$. Compute $d_1' \leftarrow \mathsf{genShare}(u, \mathsf{PIN}, n_1)$ and $d_1'' \leftarrow (d_1 - d_1') \bmod \phi(n_1)$. Send the state of the server to $\mathcal{A}$.

- If $\mathsf{c}_\mathsf{C} \geq 2$, then we move from cell $(\mathsf{c}_\mathrm{C}, \mathsf{c}_\mathrm{S}) = (2, 0)$ to cell $(2, 1)$ in Table 1, or from cell $\mathsf{c}_\mathrm{S} = 0$ to cell 1 in Table 3. All computations of the values in the internal state of the simulator have already been done. Send the state of the server to $\mathcal{A}$.

5.4.2 Leaking client's encrypted memory.

On command "Leak encrypted memory" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{cli}}$, ignore this query if key generation has not yet taken place. Otherwise, send $(\mathsf{corrupt\text{-}client}, 1)$ to $\mathcal{F}^{\mathsf{gSpl}}$ and proceed as follows:

- If $\mathsf{c}_\mathsf{S} = 0$, then we move from cell $(\mathsf{c}_\mathrm{C}, \mathsf{c}_\mathrm{S}) = (0, 0)$ to cell $(1, 0)$ in Table 1. Generate the random value u. Send the values in client's encrypted memory (the simulator has now generated all of them) to $\mathcal{A}$.

- If $\mathsf{c_S} = 1$, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 1)$ to cell $(1, 1)$ and then immediately to cell $(2, 1)$ in Table 1. Alternatively, we move from cell $\mathsf{c_C} = 0$ to cell 1 and then to cell 2 in Table 2. The functionality $\mathcal{F}^{\mathsf{gSpl}}$ sets $\mathsf{c_C} := 2$ and the simulator receives $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{gSpl}}$. Using that PIN, compute u from the already computed value d'_1. Send the values in client's encrypted memory to $\mathcal{A}$. (The adversary itself will find the values in client's unencrypted memory)

5.4.3 Leaking client's unencrypted memory with no previous leaks.

On command "Leak unencrypted memory" from $\mathcal{A}$, presumably to $\mathcal{M}_{\text{cli}}$, with $\mathsf{c_C} = 0$, ignore this query if key generation has not yet taken place. Otherwise, send $(\mathsf{corrupt\text{-}client}, 2)$ to $\mathcal{F}^{\mathsf{gSpl}}$. The functionality $\mathcal{F}^{\mathsf{gSpl}}$ sets $\mathsf{c_C} := 2$ and the simulator receives $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{gSpl}}$. Proceed as follows:

- If server has not been corrupted, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 0)$ to the cell $(2, 0)$ in Table 1. The simulator generates random u and uses PIN to construct the value d'_1 from it. Send client's unencrypted memory and the value d'_1 to the adversary.

- If server has been corrupted, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 1)$ to the cell $(2, 1)$ in Table 1. Use the PIN to compute u from the previously initialized d'_1 by inverting PRF_{n_1}. Send $\langle d'_1, u \rangle$ to the $\mathcal{A}$.

5.4.4 Leaking client's unencrypted memory, when encrypted memory has already leaked.

On command "Leak unencrypted memory" from $\mathcal{A}$, presumably to $\mathcal{M}_{\text{cli}}$, with $\mathsf{c_C} = 0$, the keys have already been generated, and the server has not been corrupted. Send $(\mathsf{corrupt\text{-}client}, 2)$ to $\mathcal{F}^{\mathsf{gSpl}}$. The functionality $\mathcal{F}^{\mathsf{gSpl}}$ sets $\mathsf{c_C} := 2$ and the simulator receives $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{gSpl}}$. We move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (1, 0)$ to the cell $(2, 0)$ in Table 1. Using the PIN, compute d'_1 from u, and send it to the adversary.

5.4.5 Fully corrupting the client.

On command "Corrupt client" from $\mathcal{A}$, presumably to $\mathcal{M}_{\text{cli}}$, send $(\mathsf{corrupt\text{-}client}, 3)$ to $\mathcal{F}^{\mathsf{gSpl}}$. If $\mathsf{c_C} \leq 1$ before that command, then compute the state of the client and send it to the adversary in the same manner as in leaking client's unencrypted memory. In any case, set $\mathsf{c_C} := 3$ and expect $\mathcal{M}_{\text{cli}}$ to be commanded by $\mathcal{A}$ in the rest of the execution.

5.5 Signing and Verification

5.5.1 Signing by client and server.

This corresponds to the client (part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ to the system (either to $\mathcal{M}_{\text{cli}}$ or $\mathcal{F}^{\mathsf{gSpl}}$) and the server (part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid})$ to the system (either to $\mathcal{M}_{\text{srv}}$ or $\mathcal{F}^{\mathsf{gSpl}}$). The simulator's next steps depend on whether the client and the server have been corrupted (i.e. the control given to the adversary) or not.

Neither controlled by the adversary. The simulator learns of this run at the moment when it receives $(\mathsf{sign\text{-}init}, \mathsf{sid}, M)$ from $\mathcal{F}^{\mathsf{gSpl}}$. The simulator uses n_1, n_2, d_1, d_2 to compute the signature σ. It sends $(\mathsf{signature}, \mathsf{sid}, \sigma)$ back the $\mathcal{F}^{\mathsf{gSpl}}$.

Client controlled by the adversary. The simulator has received the forwarded $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ command from $\mathcal{F}^{\mathsf{gSpl}}$. The simulator tells $\mathcal{A}$ that client has sent this command to $\mathcal{M}_{\text{cli}}$. The simulator participates in the signing protocol in Figure 13 with the adversary, using the values n_2 and d_2 (that it knows) to construct the second half of the signature. In the end, the simulator learns from $\mathcal{A}$ what $\mathcal{M}_{\text{cli}}$ should return to the client. The simulator tells $\mathcal{F}^{\mathsf{gSpl}}$ to return this value to the client.

Server controlled by the adversary. The simulator has received the forwarded $(\mathsf{sign}, \mathsf{sid})$ command from the ideal functionality, and also $(\mathsf{sign\text{-}init}, \mathsf{sid}, M)$ as the message from $\mathcal{F}^{\mathsf{gSpl}}$. The simulator has the values n_1 and d_1' that it needs to participate in a signing protocol with the adversary. The protocol results in a signature σ; the simulator returns $(\mathsf{signature}, \mathsf{sid}, \sigma)$ to $\mathcal{F}^{\mathsf{gSpl}}$.

Both controlled by the adversary. In this case, the simulator forwards the forwarded messages to the adversary, and learns from it the outputs that it tells $\mathcal{F}^{\mathsf{gSpl}}$ to give to the client and the server.

5.5.2 Signing by client and adversary.

This corresponds to the client (part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ to the system (either to $\mathcal{M}_{\text{cli}}$ or $\mathcal{F}^{\mathsf{gSpl}}$), and the adversary $\mathcal{A}$ telling the corrupted $\mathcal{M}_{\text{srv}}$ to run the signing protocol. In executions with $\mathcal{F}^{\mathsf{gSpl}}$, the latter is received by the simulator who sends the command $(\mathsf{sign\text{-}server}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{gSpl}}$.

The ideal functionality sends $(\mathsf{sign\text{-}init}, \mathsf{sid}, M)$ to the simulator. If the client is not under adversary's control ($c_{\text{C}} < 3$), then the simulator has the values n_1 and d_1' that it needs to run the signing protocol with the adversary. The protocol results in a signature σ; the simulator returns $(\mathsf{signature}, \mathsf{sid}, \sigma)$ to $\mathcal{F}^{\mathsf{gSpl}}$. If the client is also adversarially controlled, then $\mathcal{A}$ will tell what $\mathcal{F}^{\mathsf{gSpl}}$ returns to the client.

5.5.3 Signing by the adversary and the server.

This corresponds to the server (a part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid})$ to $\mathcal{M}_{\text{srv}}$ or $\mathcal{F}^{\mathsf{gSpl}}$, while the adversary $\mathcal{A}$, masquerading as the client, contacts $\mathcal{M}_{\text{srv}}$ (or the simulator), initiating a signing session (Figure 13) by sending it the message $\langle y, M, w' \rangle$. In this setting, the server is not corrupted, otherwise the adversary would not need to masquerade to it. For the same reason, the client is not fully corrupted ($\mathsf{c}_{\text{C}} < 3$).

If $\mathsf{c}_{\text{C}} = 0$, then the simulator sends $(\mathsf{sign\text{-}client}, M, \mathsf{PIN}_0)$ to $\mathcal{F}^{\mathsf{gSpl}}$, where PIN_0 is some constant PIN. At this point, the execution stops, because $\mathcal{F}^{\mathsf{gSpl}}$ will ignore this query. The same would have happened in the execution of the real system $\Pi^{\mathsf{4RSASplit}}$, because the adversary would not have any information about d_1'.

If $\mathsf{c}_{\text{C}} \geq 1$, then the simulator has the value u. The simulator tries out all possible values of $\mathsf{PIN}' \in \{0, \ldots, L-1\}$, and finds the one for which $M^{\mathsf{genShare}(u, \mathsf{PIN}', n_1)} \equiv y \pmod{n_1}$. The simulator sends $(\mathsf{sign\text{-}client}, M, \mathsf{PIN}')$ to $\mathcal{F}^{\mathsf{gSpl}}$. If $\mathsf{PIN}' \neq \mathsf{PIN}$ then the execution again ends, this is the same effect as in the execution of the real system $\Pi^{\mathsf{4RSASplit}}$.

If $\mathsf{PIN}' = \mathsf{PIN}$, then $\mathsf{c}_{\text{C}} := 2$. The ideal functionality sends $(\mathsf{sign\text{-}init}, \mathsf{sid}, M)$ to the simulator. The simulator completes the run of signing protocol with the adversary, obtaining the signature σ, which it sends back to the ideal functionality.

5.5.4 Verification.

The verification of signatures does not involve the simulator. However, we have to argue that the real system *Sys* (that always executes $\mathsf{Ver}(\mathsf{pk}, M, \sigma)$ at this point) returns the same verification outcomes as the ideal functionality (that also uses the records of verification

results in the internal state of $\mathcal{F}^{\mathsf{gSpl}}$). We see that due to the 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 $(M, \sigma', 1)$ has been recorded for any σ' and $b_{\text{keyleak}} = 0$. This corresponds to the construction of the pair (M, σ) without M being an argument to some signing query, and without leaking the private key. In this case, we can combine the simulator, $\mathcal{A}$, and $\mathcal{Z}$ to a challenger that breaks the UF-CMA security of RSA with the given padding. The challenger would work identically to $Sim \| \mathcal{Z} \| \mathcal{A}$, except that

- At the start of the execution, the challenger gets a RSA public key N from the environment. As long as neither party is corrupted, N will be the RSA modulus for one of them, i.e. $n_i \leftarrow N$, but the challenger does not yet know whether $i = 1$ or $i = 2$. During the key generation, the challenger generates the modulus n_{3-i} and the private exponent d_{3-i}, still not knowing what is the value of i. During signing, the CRT can be used without knowing the value of i, as long as both parties are uncorrupted. Once the corruptions start, the value i will be determined.
- At key generation, the challenger does not create the RSA modulus and the private exponent for the uncorrupted party. Instead, it will use N as the modulus (i.e. $n_1 := N$ if client stays uncorrupted, and $n_2 := N$ if server stays uncorrupted. If the client stays uncorrupted, then the challenger picks a random d_1'' as server's share of d_1''.
- During signing, if server stays uncorrupted, then the challenger calls $\mathcal{O}_{\text{sign}}(M)$ to both determine the padding for M and obtain $(\mathsf{Pad}(M))^{d_2} \bmod n_2$. If the client stays uncorrupted, then the challenger calls $\mathcal{O}_{\text{sign}}(M)$ to obtain $\mathsf{Pad}(M)$ and $\hat{\sigma}(\mathsf{Pad}(M))^{d_1} \bmod n_1$. In order to find $y = (\mathsf{Pad}(M))^{d_1'}$ as required in certain steps of the simulation, the challenger computes $\hat{\sigma} \cdot (\mathsf{Pad}(M))^{-d_1''} \bmod n_1$.

5.6 Proof summary

The simulator constructed in the proof considers different corruption scenarios – uncorrupted parties, corrupted phone, corrupted server, or phone's memory leakage. The only point, where adversary can distinguish real protocol execution from the simulated one arises during the signature verification process. The real protocol may accept a signature that the ideal functionality would reject if it has not previously recorded it. However, this would imply existence of forger that breaks the UF-CMA security of RSA. Therefore, any non-negligible distinguishing advantage would contradict the assumed UF-CMA security of underlying RSA signature scheme.

6 Server-Supported ECDSA Signatures

In this section, we show practical relevance of $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ functionality. First, we define a server-supported version of Xue et al. [XAX+21] two-party ECDSA signing. We prove proposed protocol is a secure realization of $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ in UC model (Theorem 8). One could show realization of $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ by other distributed ECDSA protocols, which we leave for future work. One *cannot* naturally realize $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ in $\mathcal{F}_{\mathsf{ECDSA}}$-hybrid model, because there would be no satisfactory way to model memory leaks.

6.1 Key Generation

Before key generation is executed 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

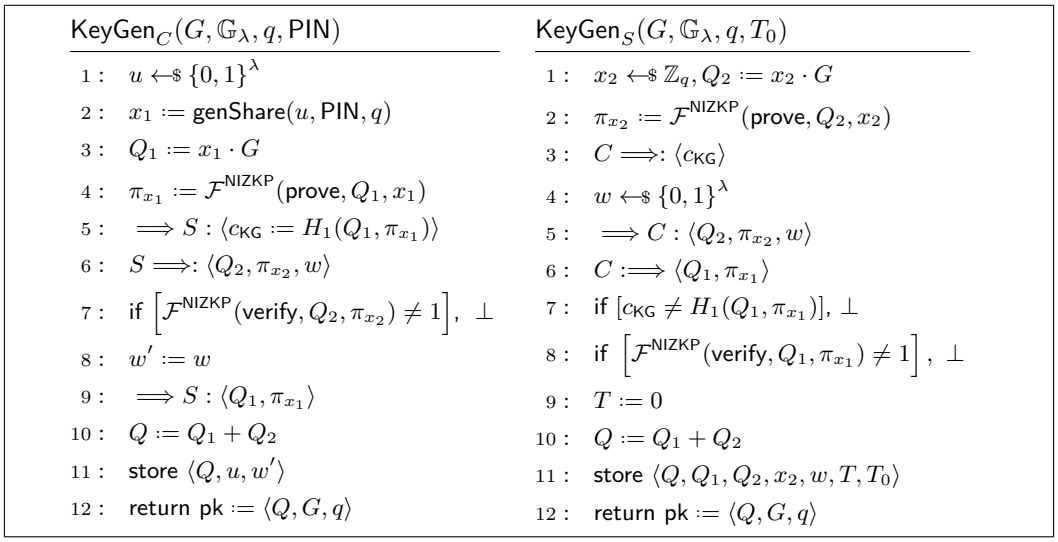

Figure 14: Server-supported ECDSA key generation

implementations of $\mathcal{F}^{\mathsf{rpoRO}}$ with output length 2λ. Parameters for $\mathcal{F}^{\mathsf{MtA}}$ and $\mathcal{F}^{\mathsf{NIZKP}}$ for both client and server are properly defined.

First, client C samples a random bitstring $u \leftarrow\$ \{0,1\}^\lambda$ to generate their secret key share x_1 derived from user-provided PIN using $\mathsf{genShare}(\cdot)$ function. Then C computes corresponding public key share $Q_1 := x_1 \cdot G$ and produce π_{x_1}, a proof of knowledge of x_1. With the first sent message, C commits to values $c_{\mathsf{KG}} := \langle H_1(Q_1, Q_1', x'', \pi_{x_1})\rangle$.

On their side, server S generates its share of the secret and public key shares x_2, $Q_2 := x_2 \cdot G$ and produces a proof π_{x_2}. Next, S samples clone detection string $w \leftarrow\$ \{0,1\}^\lambda$. S sends Q_2, w, π_{x_2} to the client.

Client verifies π_{x_2}, stores bitstring $w' := w$ and sends opening to the commitment c_{KG}. Upon receiving opening, server verifies corresponding commitment and validity of the proof π_{x_1} against Q_1. Server sets counter $T := 0$ to track number of incorrect PIN guesses made by the client.

Finally, both client and server calculate resulting public key $\mathsf{pk} := \langle Q := Q_1 + Q_2, G, q\rangle$. Client stores only Q, u, w' values and must securely delete all the remaining values received or generated during the key generation protocol. They do so to prevent potential PIN guesses during memory leakage. Server stores a set values $\langle Q, Q_1, Q_2, x_2, w, T, T_0\rangle$ on their side. More detailed description of distributed key generation is presented in the Figure 14.

6.2 Signing

Client C samples a ephemeral key $k_1 \leftarrow\$ \mathbb{Z}_q$, computes the curve elliptic point $R_1 = k_1 \cdot G$ and corresponding proof of knowledge π_{k_1}. In parallel, server computes their ephemeral shares $k_2 \leftarrow\$ \mathbb{Z}_q$, $R_2 = k_2 \cdot G$ and a proof π_{k_2}. Since we want to authenticate any client after their message, client additionally samples $x_1 := \mathsf{genShare}(u, \mathsf{PIN}, q)$ using user provided PIN and produce a proof π_{x_1}. Then, C sends $\langle c_{\mathsf{sig}} := H_2(R_1, w, M, \pi_{x_1}, \pi_{k_1}), \pi_{x_1}, w\rangle$ to the server. S checks the validity of π_{x_1} against locally stored public key share Q_1, restarting the signing protocol upon failure. If $T = T_0$ or $w' \neq w$, server does not participate in any future communication with the client.

Upon success, both server and client run MtA protocol with inputs k_1 and x_2^* to receive correlated outputs t_c and t_s. Next, S proceeds with calculating values for consistency check by generating $y \leftarrow\$ \mathbb{Z}_q$, computing $\mathsf{hid} := t_s + x_2^* \cdot y - x_2$ and sending $\langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2}\rangle$ to C. On their side, C runs a consistency check that $(t_c + \mathsf{hid}) \cdot G = (y + k_1) \cdot Q_2^* - (Q - Q_1)$

$\mathsf{Sign}_C(M, \mathsf{pk}, \mathsf{PIN}, u, w')$	$\mathsf{Sign}_S(\mathsf{pk}, x_2, w, T)$
1 : $k_1 \leftarrow\$ \mathbb{Z}_q, R_1 := k_1 \cdot G$	1 : $k_2 \leftarrow\$ \mathbb{Z}_q, R_2 := k_2 \cdot G$
2 : $\pi_{k_1} := \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_1, k_1)$	2 : $\pi_{k_2} := \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, R_2, k_2)$
3 : $x_1 := \mathsf{genShare}(u, \mathsf{PIN}, q), Q_1 := x_1 \cdot G$	3 : $C \Longrightarrow: \langle c_{\mathsf{sig}}, \pi_{x_1}, w \rangle$
4 : $\pi_{x_1} := \mathcal{F}^{\mathsf{NIZKP}}(\mathsf{prove}, Q_1, x_1)$	4 : if $\left[w' \neq w \vee T \geq T_0\right]$, $\perp$
5 : $c_{\mathsf{sig}} := H_2(R_1, w, M, \pi_{x_1}, \pi_{k_1})$	5 : if $\left[\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, Q_1, \pi_{x_1}) \neq 1\right]$, RESTART
6 : $\Longrightarrow S : \langle M, c_{\mathsf{sig}}, \pi_{x_1}, w \rangle$	6 : $x_2^* \leftarrow\$ \mathbb{Z}_q, Q_2^* := x_2^* \cdot G$
7 : $t_c := \mathcal{F}^{\mathsf{MtA}}(\mathsf{input}, k_1)$	7 : $t_s := \mathcal{F}^{\mathsf{MtA}}(\mathsf{input}, x_2^*)$
8 : $S \Longrightarrow: \langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2} \rangle$	8 : $y \leftarrow\$ \mathbb{Z}_q$
9 : $\mathsf{ch} := (t_c + \mathsf{hid}) \cdot G$	9 : $\mathsf{hid} := t_s + x_2^* \cdot y - x_2 \mod q$
10 : if $[\mathsf{ch} \neq (y + k_1) \cdot Q_2^* - (Q - Q_1)]$, $\perp$	10 : $\Longrightarrow C : \langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2} \rangle$
11 : if $\left[\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, R_2, \pi_{k_2}) \neq 1\right]$, $\perp$	11 : $C :\Longrightarrow \langle R_1, s_1, \pi_{k_1} \rangle$
12 : $R := (y + k_1) \cdot R_2, r := R_x \mod q$	12 : if $\langle c_{\mathsf{sig}} \neq H_2(R_1, w, M, \pi_{x_1}, \pi_{k_1}) \rangle$, $\perp$
13 : $x_1^* := x_1 - (t_c + \mathsf{hid}) \mod q$	13 : if $\left[\mathcal{F}^{\mathsf{NIZKP}}(\mathsf{verify}, R_1, \pi_{k_1}) \neq 1\right]$, $\perp$
14 : $s_1 := (k_1 + y)^{-1}(H_0(M) + r x_1^*) \mod q$	14 : $R := k_2 \cdot R_1 + k_2 \cdot y \cdot G, r := R_x \mod q$
15 : $\Longrightarrow S : \langle R_1, s_1, \pi_{k_1} \rangle$	15 : $s := k_2^{-1}(s_1 + r \cdot x_2^*) \mod q$
16 : $S \Longrightarrow: \langle r, s, w \rangle$	16 : if $[\mathsf{ECDSAVerify}(\mathsf{pk}, M, \langle r, s \rangle) \neq 1]$, $\perp$
17 : if $[\mathsf{ECDSAVerify}(\mathsf{pk}, M, \langle r, s \rangle) \neq 1]$, $\perp$	17 : $w \leftarrow\$ \{0,1\}^\lambda$
18 : $w' := w$	18 : $T := 0$
19 : return $\sigma := \langle r, s \rangle$	19 : $\Longrightarrow C : \langle r, s, w \rangle$
	20 : return $\sigma := \langle r, s \rangle$

Figure 15: Server-supported ECDSA signing

together with verifying the proof π_{k_2}. If they are successful, C calculates a partial signature $s_1 := (k_1 + y)^{-1}(H_0(M) + r x_1^*) \mod q$ and sends it to S with π_{k_1}.

If an opening of the commitment c_{sig} and proof π_{k_1} is valid, S computes final signature $s := k_2^{-1}(s_1 + r \cdot x_2^*) \mod q$ using partial signature s_1 supplied by C. After verifying signature $\langle r, s \rangle$, server generates new clone detection bitstring $w \leftarrow\$ \{0,1\}^\lambda$ and sends in to C. The formal description of server-supported ECDSA signing protocol is presented in Figure 15.

7 Security of Server-Supported ECDSA

Theorem 8. *Server-supported ECDSA signing presented in Figures 14 and 15 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.*

In order to show that Π^{ECDSA} is at least as secure as $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$, we will construct a simulator *Sim*, such that for any adversary $\mathcal{A}$, no environment $\mathcal{Z}$ can distinguish an execution with $\Pi^{\mathsf{ECDSA}} \| \mathcal{A} \| \mathcal{F}^{\mathsf{rpoRO}}$ from an execution with $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}} \| Sim \| \mathcal{A} \| \mathcal{F}^{\mathsf{rpoRO}}$. The state of the simulator contains the values mentioned in the description of the ideal functionality and the protocols, namely:

$$\mathsf{c_S}, \mathsf{c_C}, Q, Q_1, Q_2, x, x_1, x_2, u, T, \mathsf{PIN}, w, \tag{1}$$

Table 4: The values that are defined in the simulator, if both the server and the client were uncorrupted during key generation. For each $(\mathsf{c_C}, \mathsf{c_S})$ value, the cells to the left and below are accounted for, too.

	$\mathsf{c_S} = 0$	$\mathsf{c_S} = 1$
$\mathsf{c_C} \geq 2$	PIN, x_1, Q_1, Q_2	x
$\mathsf{c_C} = 1$	u	N/A. Go to $\mathsf{c_C} \geq 2$
$\mathsf{c_C} = 0$	Q, T, w	Q_1, Q_2, x_2

Table 5: The values that are defined in the simulator, if the server was corrupted during key generation

$\mathsf{c_C} \geq 2$	PIN, u, x, x_1
$\mathsf{c_C} = 1$	NA. Go to $\mathsf{c_C} \geq 2$
$\mathsf{c_C} = 0$	Q, Q_1, Q_2, T, w, x_2

where $x = x_1 + x_2$ is the full private key. The state of *Sim* also contains the internal states of $\mathcal{F}^{\mathsf{NIZKP}}$ and $\mathcal{F}^{\mathsf{MtA}}$. Most of these values are undefined at the start of an execution. Actually, only $\mathsf{c_C} := 0$ and $\mathsf{c_S} := 0$ are defined at the beginning, and the internal states of the simulated ideal functionalities are initialized at the beginning of the execution. These internal states evolve during the execution. Other values in (1) become defined while simulating key generation and also when the corruption level of the client or the server increases. Tables 4–6 give an overview of when each of these other values is initialized. In these tables, the values in the left / bottom cell are initialized during key generation. The rest are initialized when the corruption level of the client or the server increases. Recall that the case $\mathsf{c_C} = \mathsf{c_S} = 1$ is impossible.

Below, we outline the manner in which the simulator interacts with and reacts to commands from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ and $\mathcal{A}$.

7.1 Initialization

During the initialization, simulator receives from $\mathcal{A}$ the descriptions of the algorithms of $\mathcal{F}^{\mathsf{NIZKP}}$ and sets $T = 0$. It also marks $\mathcal{F}^{\mathsf{MtA}}$ as initialized.

7.2 Corruption before key generation

On command "Corrupt server" from $\mathcal{A}$ before the key generation has started, send (corrupt-server) to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$.

On command "Corrupt client" from $\mathcal{A}$ before the key generation has started, send (corrupt-client, 3) to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$.

7.3 Key generation

On command (keygen-start, Q) from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ (here $Q \in \mathbb{G}$ is the public key), the simulator saves Q. It proceeds as follows; depending on the corruption level of the client and server.

Table 6: The values that are defined in the simulator, if the client was corrupted during key generation

$\mathsf{c_S} = 0$	$\mathsf{c_S} = 1$
Q, Q_1, Q_2, T, PIN, w, x_1	x, x_2

7.3.1 Neither is corrupted.

Simulate the execution of both client and server inside *Sim*, running the distributed key generation protocol in Fig. 14. During the execution, initialize only the values named in the cell $\mathsf{c_P} = \mathsf{c_S} = 0$ in table 4; the outputs of the key generation protocol can be computed from these values. In this case, the real protocol execution is identical to the simulated one, since all values are generated in the same way as in the real protocol. Send (keygen-ok) back to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

7.3.2 Client is corrupted.

Simulate the server's execution of the protocol, defining the values in the "$\mathsf{c_S} = 0$" cell of Table 6. I.e. generate x_2, and w, compute $Q_2 = x_2 \cdot G$ and $Q_1 = Q - Q_2$, use Sim-Setup and Sim-Prove to create π_{x_2}. Upon receiving c_{KG} from $\mathcal{A}$ (as if coming from the corrupted client), send back Q_2, π_{x_2}, w, receive Q_1, π_{x_1}. Run the verification check of (Q_1, π_{x_1}), using the algorithms Verify, Extract as necessary (see App. 7.7). If the verification fails, then send (keygen-stop) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Otherwise extract x_1 from π_{x_1}, set $T := 0$, store Q_1, Q_2, x_2, w, T, send (keygen-ok) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

The key generation with corrupted client triggers the leakage of client's PIN. Hence receive (corrupt-pin, PIN) from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

The real protocol execution is a bit different than the simulated one: in real protocol we define $Q := Q_1 + Q_2$, while in simulated protocol we define $Q_1 := Q - Q_2$. Still, the distributions of all values are the same.

7.3.3 Server is corrupted.

Simulate client's execution of the protocol, defining the values in the "$\mathsf{c_C} = 0$" cell of Table 5. Generate random c_{KG}, send c_{KG} to $\mathcal{A}$ (as if going to the server). Upon receiving (Q_2, π_{x_2}, w) from $\mathcal{A}$, compute $Q_1 := Q - Q_2$. Simulate the proof π_{x_1}. Program $\mathcal{F}^{\mathsf{rpoRO}}$, such that $c_{\mathsf{KG}} = H_1(Q_1, \pi_{x_1})$. Verify the proof π_{x_2}. If it does not verify, then send (keygen-stop) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. If it verifies then extract x_2 from π_{x_2} and send (Q, π_{x_1}) to $\mathcal{A}$.

Again the real protocol execution is a bit different than the simulated one, but not recognizably so.

7.3.4 Both parties are corrupted.

In this case, the adversary $\mathcal{A}$ runs the key generation protocol. At the end of the protocol, the adversary tells *Sim* what the machines $\mathcal{M}_{\mathrm{cli}}$ and $\mathcal{M}_{\mathrm{srv}}$ should output to the environment. The simulator forwards these messages to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. The simulator also receives PIN and x from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

7.4 Corruption after key generation

7.4.1 Corrupting the server.

On command "Corrupt server" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{srv}}$, after the keys have been generated, send (corrupt-server) to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ and proceed as follows:

- If $\mathsf{c_C} = 0$, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 0)$ to cell $(0, 1)$ in Table 4. Generate random x_2, compute $Q_2 := x_2 \cdot G$ and $Q_1 := Q - Q_2$. Send server's state (Fig. 14) to $\mathcal{A}$.
- If $\mathsf{c_C} = 1$, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (1, 0)$ to cell $(2, 1)$ in Table 4. I.e., $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ sets $\mathsf{c_C} := 2$ and the simulator receives (corrupt-pin, PIN) and (private-key, x) from

$\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. Compute $x_1 := \mathsf{genShare}(u, \mathsf{PIN})$, $x_2 := x - x_1$, $Q_1 := x_1 \cdot G$, $Q_2 := x_2 \cdot G$. Send the state of the server to $\mathcal{A}$.

- If $\mathsf{c_C} \geq 2$ then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (2, 0)$ to cell $(2, 1)$ in Table 4, or from cell "$\mathsf{c}_S = 0$" to cell "$\mathsf{c}_S = 1$" in Table 6. The simulator receives $(\mathsf{private\text{-}key}, x)$ from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ and computes $x_2 := x - x_1$. It can now send the state of the server to $\mathcal{A}$.

7.4.2 Leaking client's encrypted memory.

On command "Leak encrypted memory" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{cli}}$, ignore this query if key generation has not yet taken place. Otherwise, send $(\mathsf{corrupt\text{-}client}, 1)$ to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ and proceed as follows:

- If $\mathsf{c_S} = 0$, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 0)$ to cell $(1, 0)$ in Table 4. Generate the random value u. Send the values in client's encrypted memory (the simulator has now generated all of them) to $\mathcal{A}$.
- If $\mathsf{c_S} = 1$, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 1)$ to cell $(1, 1)$ and then immediately to cell $(2, 1)$ in Table 4. Alternatively, we move from cell $\mathsf{c_C} = 0$ to cell 1 and then to cell 2 in Table 5. The functionality $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ sets $\mathsf{c_C} := 2$ and the simulator receives both $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ and $(\mathsf{private\text{-}key}, x)$ from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. Compute $x_1 := x - x_2$. Using PIN, compute u from x_1. Send the values in client's encrypted memory to $\mathcal{A}$. (The adversary itself will find the values in client's unencrypted memory).

7.4.3 Leaking client's unencrypted memory with no previous leaks.

On command "Leak unencrypted memory" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{cli}}$, with $\mathsf{c_C} = 0$, ignore this query if key generation has not yet taken place. Otherwise, send $(\mathsf{corrupt\text{-}client}, 2)$ to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. The functionality $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ sets $\mathsf{c_C} := 2$ and the simulator receives $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. Proceed as follows:

- If server has not been corrupted, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 0)$ to the cell $(2, 0)$ in Table 4. The simulator generates random u and uses PIN to construct the value x_1 from it. It computes $Q_1 := x_1 \cdot G$ and $Q_2 := Q - Q_1$. Send client's unencrypted memory and the value x_1 to the adversary.
- If server has been corrupted, then we move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (0, 1)$ to the cell $(2, 1)$ in Table 1. Simulator receives $(\mathsf{private\text{-}key}, x)$ from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. Compute $x_1 := x - x_2$. Use the PIN to compute u from x_1 by inverting $\mathsf{genShare}(\cdot, \cdot, q)$. Send $\langle x_1, u \rangle$ to $\mathcal{A}$.

7.4.4 Leaking client's unencrypted memory, when encrypted memory has already leaked.

On command "Leak unencrypted memory" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{cli}}$, with $\mathsf{c_C} = 0$, the keys have already been generated, and the server has not been corrupted. Send $(\mathsf{corrupt\text{-}client}, 2)$ to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. The functionality $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ sets $\mathsf{c_C} := 2$ and the simulator receives $(\mathsf{corrupt\text{-}pin}, \mathsf{PIN})$ from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. We move from cell $(\mathsf{c_C}, \mathsf{c_S}) = (1, 0)$ to the cell $(2, 0)$ in Table 4. Using the PIN, compute x_1 from u, and send it to the adversary. Also compute $Q_1 := x_1 \cdot G$ and $Q_2 := Q - Q_1$.

7.4.5 Fully corrupting the client.

On command "Corrupt client" from $\mathcal{A}$, presumably to $\mathcal{M}_{\mathrm{cli}}$, send $(\mathsf{corrupt\text{-}client}, 3)$ to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. If $\mathsf{c_C} \leq 1$ before that command, then compute the state of the client and send it to the adversary in the same manner as in leaking client's unencrypted memory. In any case, set $\mathsf{c_C} := 3$ and expect $\mathcal{M}_{\mathrm{cli}}$ to be commanded by $\mathcal{A}$ in the rest of the execution.

7.5 Signing and verification

7.5.1 Signing by client and server.

This corresponds to the client (part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ to the system (either to $\mathcal{M}_{\text{cli}}$ or $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$) and the server (part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid})$ to the system (either to $\mathcal{M}_{\text{srv}}$ or $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$). The simulator's next steps depend on whether the client and the server have been corrupted (i.e. the control given to the adversary) or not.

Neither controlled by the adversary. The simulator learns of this run at the moment when it receives $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. The simulator responds with $(\mathsf{sign\text{-}proceed}, \mathsf{sid})$.

Client controlled by the adversary. The simulator has received the forwarded $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ command from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. The simulator tells $\mathcal{A}$ that client has sent this command to $\mathcal{M}_{\text{cli}}$. The simulator also receives $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. The simulator runs the signing protocol in Fig. 15 with the adversary, with the simulator playing the role of server, and the result of the signing being $\sigma = \langle r, s \rangle$. Let $R \in \mathbb{G}$ be a point with the x-coordinate r. From simulator's point of view, this run proceeds as follows:

1. Receive $M, c_{\mathsf{sig}}, \pi_{x_1}, w$ from $\mathcal{A}$.
2. If w is not equal to the stored w or π_{x_1} does not verify, then send $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Otherwise:
3. Send $(\mathsf{observe}, \mathsf{sid})$ to $\mathcal{F}^{\mathsf{rpoRO}}$. Receive the list Q_{sid}, find from it the H_2-query that produced c_{sig}. Find R_1, π_{k_1}. Extract k_1 from R_1.
4. Receive $(\mathsf{input}, \mathsf{sid}, k_1)$ from $\mathcal{A}$, presumably meant for $\mathcal{F}^{\mathsf{MtA}}$. If this k_1 is different from the previous one, then update it, but also remember to return $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ at the end of the simulation.
5. Generate random $y, t_c, m \in \mathbb{Z}_q$. Define $n := y + k_1$, $R_2 := n^{-1} \cdot R$, $\mathsf{hid} := n \cdot m - t_c$, $Q_2^* := n^{-1} \cdot Q_2 + m \cdot G$. Fake the proof π_{k_2}.
6. Send $(\mathsf{output}, \mathsf{sid}, t_c)$ to $\mathcal{A}$, presumably coming from $\mathcal{F}^{\mathsf{MtA}}$.
7. Send the message $\langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2} \rangle$ to $\mathcal{A}$, i.e. to the corrupted client.
8. Receive $\langle R_1, s_1, \pi_{k_1} \rangle$ from $\mathcal{A}$.
9. If $c_{\mathsf{sig}} \neq H_2(R_1, w, M, \pi_{x_1}, \pi_{k,1})$ or the proof π_{k_1} does not verify, then send $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Otherwise:
10. Generate new w. Send $\langle r, s, w \rangle$ to $\mathcal{A}$ and $(\mathsf{sign\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

In this way, the values seen by the adversary (in messages that the server sends to the client, or the client obtains from ideal functionalities) are distributed as expected. Indeed,

- t_c and y should be uniformly randomly distributed, and independent from previous values;
- R_2 should be a uniformly random element of $\mathbb{G}$. It is indeed, because n (and n^{-1}) is uniformly random.
- Q_2^* should also be a uniformly random element of $\mathbb{G}$. It is, because m is uniformly random.
- hid should be such, that the equality check involving ch passes. It does indeed pass.

Server controlled by the adversary. The simulator has received the forwarded $(\mathsf{sign}, \mathsf{sid})$ command from the ideal functionality, and also $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ as the message from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. The simulator runs the signing protocol in Fig. 15 with the adversary, with the simulator playing the rol of client, and the result of the signing being $\sigma = \langle r, s \rangle$. Let $R \in \mathbb{G}$ be a point with the x-coordinate r. From simulator's point of view, this run proceeds as follows:

1. Fake the proof π_{x_1} and generates a random c_{sig}.
2. Send $\langle M, c_{\mathsf{sig}}, \pi_{x_1}, w \rangle$ to $\mathcal{A}$ (i.e. to corrupted server).
3. Receive $(\mathsf{input}, \mathsf{sid}, x_2^*)$ from $\mathcal{A}$, presumably meant for $\mathcal{F}^{\mathsf{MtA}}$.
4. Generate a random $t_s \leftarrow\!\!\$\ \mathbb{Z}_q$ and send $(\mathsf{output}, \mathsf{sid}, t_c)$ to $\mathcal{A}$, presumably coming from $\mathcal{F}^{\mathsf{MtA}}$.
5. Receive the message $\langle R_2, Q_2^*, y, \mathsf{hid}, \pi_{k_2} \rangle$ from $\mathcal{A}$.
6. Verify π_{k_2}. If it does not verify then finish the execution and send $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Otherwise:
7. Verify that $\mathsf{hid} - t_s = y \cdot x_2' - x_2$. If this is not the case, then finish the execution and send $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Otherwise:
8. Extract k_2 from π_{k_2}. Define $R_1 := k_2^{-1} \cdot R - y \cdot G$ and $s_1 := s \cdot k_2 - r \cdot x_2^*$. Fake the proof π_{k_1}.
9. Send $(\mathsf{program\text{-}RO}, (R_1, w, M, \pi_{x_1}, \pi_{k_1}), c_{\mathsf{sig}})$ to $\mathcal{F}^{\mathsf{rpoRO}}$.
10. Send $\langle R_1, s_1, \pi_{k_1} \rangle$ to $\mathcal{A}$.
11. Receive $\langle r, s, w \rangle$ from $\mathcal{A}$, where w is the next clone detection string. Update it. If $\langle r, s \rangle = \sigma$ then send $(\mathsf{sign\text{-}ok}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$. Otherwise send $(\mathsf{sign\text{-}fail}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

In this way, the values seen by the adversary (in messages that the client sends to the server, or the server obtains from ideal functionalities) are distributed as expected. Indeed,

- t_s is uniformly random, as expected.
- R_1 and s_1 are such, that they lead to the computation of the signature $\sigma = \langle r, s \rangle$ that the simulator received from $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.
- The result of the equality test is the same as the result of the test in $\mathsf{Sign}_{\mathrm{C}}$.

Both controlled by the adversary. In this case, the simulator forwards the forwarded messages to the adversary, and learns from it the outputs that it tells $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ to give to the client and the server.

7.5.2 Signing by client and adversary.

This corresponds to the client (part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ to the system (either to $\mathcal{M}_{\mathrm{cli}}$ or $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$), and the adversary $\mathcal{A}$ telling the corrupted $\mathcal{M}_{\mathrm{srv}}$ to run the signing protocol. In executions with $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$, the latter is received by the simulator who sends the command $(\mathsf{sign\text{-}server}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$.

The ideal functionality sends $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ to the simulator. If the client is not under adversary's control ($c_{\mathrm{C}} < 3$), then the simulator can run the signing protocol with the adversary in the same manner as described above ("Server controlled by the adversary"). If the client is also adversarially controlled, then $\mathcal{A}$ will tell what $\mathcal{F}_{\mathsf{ECDSA}}^{\mathsf{Spl}}$ returns to the client.

7.5.3 Signing by the adversary and the server.

This corresponds to the server (a part of $\mathcal{Z}$) sending the command $(\mathsf{sign}, \mathsf{sid})$ to $\mathcal{M}_{\text{srv}}$ or $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$, while the adversary $\mathcal{A}$, masquerading as the client, contacts $\mathcal{M}_{\text{srv}}$ (or the simulator), initiating a signing session (Fig. 15) by sending it the message $\langle M, c_{\mathsf{sig}}, \pi_{x_1}, w\rangle$. In this setting, the server is not corrupted, otherwise the adversary would not need to masquerade to it. For the same reason, the client is not fully corrupted ($\mathsf{c}_{\mathrm{C}} < 3$).

If $\mathsf{c}_{\mathrm{C}} = 0$, then the simulator sends $(\mathsf{sign\text{-}client}, M, \mathsf{PIN}_0)$ to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$, where PIN_0 is some constant PIN. At this point, the execution stops, because $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$ will ignore this query. The same would have happened in the execution of the real system, because the adversary would not have any information about x_1 and Q_1.

If $\mathsf{c}_{\mathrm{C}} \geq 1$, then the simulator has the value u. The simulator tries out all possible values of $\mathsf{PIN}' \in \{0, \ldots, L-1\}$, and finds the one where π_{x_1} verifies with respect to $\mathsf{genShare}(u, \mathsf{PIN}', q)\cdot G$. The simulator sends $(\mathsf{sign\text{-}client}, M, \mathsf{PIN}')$ to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{ECDSA}}$. If $\mathsf{PIN}' \neq \mathsf{PIN}$ then the execution again ends, this is the same effect as in the execution of the real system.

If $\mathsf{PIN}' = \mathsf{PIN}$, then $\mathsf{c}_{\mathrm{C}} := 2$. The ideal functionality sends $(\mathsf{sign\text{-}ongoing}, \mathsf{sid}, M, \sigma)$ to the simulator. The simulator completes the run of signing protocol with the adversary, in the same manner as above ("Client controlled by the adversary").

7.6 Handling to queries to $\mathcal{F}^{\mathsf{MtA}}$

Adversary may make $(\mathsf{input}, \mathsf{sid}, x)$ queries towards $\mathcal{F}^{\mathsf{MtA}}$. These are expected during the signing protocol, where their handling takes place as described above. These queries may also take place at other times, with other session identifiers. These are replied as stated in Fig. 3, generating random shares of the product of inputs.

7.7 Handling queries to $\mathcal{F}^{\mathsf{NIZKP}}$

Adversary may make queries to generate or verify the proofs handled by $\mathcal{F}^{\mathsf{NIZKP}}$. These are handled normally, considering also that the simulated proofs generated by the simulator will also be reported as passing.

Simulator constructed throughout the proof uses programmability of global random oracle and the extractability/simulatability of the NIZKP functionality to generate proofs and signing transcripts on behalf of the honest parties. Since the NIZKPs are perfectly simulated and the random oracle can be programmed consistently (except with negligible probability due to collisions), the view of adversary in the simulated execution is indistinguishable from the the real protocol. We see that throughout the protocol, the simulator is able to simulate the replies to all the queries made by the adversary, or by the environment. Hence proposed server-supported ECDSA protocol securely UC-realizes the ideal functionality.

8 RSA vs. ECDSA

In this section, we compare RSA-based server-supported signing against ECDSA-based one. As could be seen in the Table 7, ECDSA clearly outperforms RSA in terms key and signature sizes. Still, the presented ECDSA protocol requires to run NIZKP and MtA. One could argue that the usage of Fischlin transform in ZKP incurs additional costs to the protocol, but as recently pointed out by Chen and Lindell [CL24] those costs are unnoticeable in practical applications. The MtA's major effect on the protocol will be either on the bandwidth (if implemented using OT [DKLS18, DKLS19]) or on the execution time (if implemented with linear homomorphic encryption [LN18, CCL+20]).

Table 7: Comparison of RSA (3072 modulus) and ECDSA server-supported signing (P−256 curve).

Protocols	Phone's Key Size	Server's Key Size	Signature Size
Server-supported RSA	3328 bits	9344 bits	6144 bits
Server-supported ECDSA (this work)	512 bits	1666 bits	512 bits

Buldas et al. [BKLO17] protocol does not require any additional subprotocols, though, key generation could be slowed by (l, s)-safe prime generation on mobile devices.

One "*hidden*" advantage of RSA-based server-supported protocol is its compatibility with standard HSMs on the supporting server side, if HSM allows to produce unpadded RSA signatures [8]. Conversely, we tend to believe that implementation of ECDSA-based server-supported protocol with HSMs presents significant challenges.

9 Conclusion and Future Work

In this paper, we introduced the ideal functionalities $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$, which capture the security and functional properties of server-supported signing. We have shown that both functionalities could be securely realized in UC with RSA and ECDSA server-supported protocols. There are several main directions of future work. First, could be showing how could one securely realize ideal functionalities $\mathcal{F}^{\mathsf{gSpl}}$ and/or $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ for other signature scheme such as EdDSA, BLS and ML-DSA. Another, defining security of the whole infrastructure for server-supported signing such as Smart-ID in UC model. This would include defining security against denial-of-service attacks, more nuanced session tracking system and modeling HSM-calls in UC. Finally, we leave implementation of server-supported ECDSA and LAN/WAN benchmarks against Buldas et al. protocol for future work.

10 Acknowledgments

This paper is the result of the research project funded by Estonian Research Council under the grant number PRG1780. We would like to thank the reviewers for their constructive and meticulous feedback.

References

[ATW97] N. Asokan, G. Tsudik, and M. Waidner. Server-supported signatures*. *Journal of Computer Security*, 5(1):91–108, 1997. doi:10.3233/JCS-1997-5105.

[BB04] Kemal Bicakci and Nazife Baykal. Server Assisted Signatures Revisited. In Tatsuaki Okamoto, editor, *Topics in Cryptology – CT-RSA 2004*, volume 2964 of *Lecture Notes in Computer Science*, pages 143–156, San Francisco, CA, USA, February 23–27, 2004. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-24660-2_12.

[BB05] Kemal Bicakci and Nazife Baykal. Improved server assisted signatures. *Computer Networks*, 47(3):351–366, 2005. URL: https://www.sciencedirect.com/science/article/pii/S138912860400235X, doi:10.1016/j.comnet.2004.08.008.

[8] https://www.smart-id.com/wordpress/wp-content/uploads/2017/01/smart-id-technical-overview-v0.6.html

[BHJL16] Fabrice Benhamouda, Javier Herranz, Marc Joye, and Benoît Libert. Efficient Cryptosystems From 2^k-th Power Residue Symbols. *Journal of Cryptology*, 30(2):519–549, April 2016. URL: http://dx.doi.org/10.1007/s00145-016-9229-5, doi:10.1007/s00145-016-9229-5.

[BHMS17] Nina Bindel, Udyani Herath, Matthew McKague, and Douglas Stebila. Transitioning to a Quantum-Resistant Public Key Infrastructure. In Tanja Lange and Tsuyoshi Takagi, editors, *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017*, pages 384–405, Utrecht, The Netherlands, June 26–28, 2017. Springer, Cham, Switzerland. doi:10.1007/978-3-319-59879-6_22.

[BKLO17] Ahto Buldas, Aivo Kalu, Peeter Laud, and Mart Oruaas. Server-supported RSA signatures for mobile devices. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, pages 315–333. Springer, 2017. doi:10.1007/978-3-319-66402-6_19.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93: 1st Conference on Computer and Communications Security*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press. doi:10.1145/168588.168596.

[BR96] Mihir Bellare and Phillip Rogaway. The exact security of digital signatures - how to sign with RSA and rabin. In Ueli M. Maurer, editor, *Advances in Cryptology - EUROCRYPT '96, International Conference on the Theory and Application of Cryptographic Techniques, Saragossa, Spain, May 12-16, 1996, Proceeding*, volume 1070 of *Lecture Notes in Computer Science*, pages 399–416. Springer, 1996. doi:10.1007/3-540-68339-9_34.

[Bra26] Luis Brando. *NIST First Call for Multi-Party Threshold Schemes*. National Institute of Standards and Technology, 2026. URL: http://dx.doi.org/10.6028/NIST.IR.8214C, doi:10.6028/nist.ir.8214c.

[Can01] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In Bob Werner, editor, *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press. doi:10.1109/SFCS.2001.959888.

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5), September 2020. doi:10.1145/3402457.

[CCL+19] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Two-Party ECDSA from Hash Proof Systems and Efficient Instantiations. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 191–221, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Cham, Switzerland. doi:10.1007/978-3-030-26954-8_7.

[CCL+20] Guilhem Castagnos, Dario Catalano, Fabien Laguillaumie, Federico Savasta, and Ida Tucker. Bandwidth-efficient threshold EC-DSA. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part II*, volume 12111 of *Lecture Notes in Computer Science*, pages 266–296,

Edinburgh, UK, May 4–7, 2020. Springer, Cham, Switzerland. doi:10.1007/978-3-030-45388-6_10.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part I*, volume 10820 of *Lecture Notes in Computer Science*, pages 280–312, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Cham, Switzerland. doi:10.1007/978-3-319-78381-9_11.

[CDG+25] Megan Chen, Pousali Dey, Chaya Ganesh, Pratyay Mukherjee, Pratik Sarkar, and Swagata Sasmal. Universally composable non-interactive zero-knowledge from sigma protocols via a new straight-line compiler. In Tibor Jager and Jiaxin Pan, editors, *PKC 2025: 28th International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 15674 of *Lecture Notes in Computer Science*, pages 381–417, Røros, Norway, May 12–15, 2025. Springer, Cham, Switzerland. doi:10.1007/978-3-031-91820-9_13.

[CEN25] Sofía Celi, Daniel Escudero, and Guilhem Niot. Share the MAYO: Thresholdizing MAYO. In Ruben Niederhagen and Markku-Juhani O. Saarinen, editors, *Post-Quantum Cryptography - 16th International Workshop, PQCrypto 2025, Part I*, pages 165–198, Taipei, Taiwan, April 08–10, 2025. Springer, Cham, Switzerland. doi:10.1007/978-3-031-86599-2_6.

[CGG+20] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1769–1787, Virtual Event, USA, November 9–13, 2020. ACM Press. doi:10.1145/3372297.3423367.

[CL15] Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from ddh. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, pages 487–505, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-16715-2_26.

[CL24] Yi-Hsiu Chen and Yehuda Lindell. Optimizing and implementing fischlin's transform for UC-secure zero knowledge. *IACR Communications in Cryptology*, 1(2), 2024. doi:10.62056/a66chey6b.

[CLNS16] Jan Camenisch, Anja Lehmann, Gregory Neven, and Kai Samelin. Virtual Smart Cards: How to Sign with a Password and a Server. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16: 10th International Conference on Security in Communication Networks*, volume 9841 of *Lecture Notes in Computer Science*, pages 353–371, Amalfi, Italy, August 31 – September 2, 2016. Springer, Cham, Switzerland. doi:10.1007/978-3-319-44618-9_19.

[DEF24] Poulami Das, Andreas Erwig, and Sebastian Faust. Shared-Custodial Password-Authenticated Deterministic Wallets. In Clemente Galdi and Duong Hieu Phan, editors, *SCN 24: 14th International Conference on Security in Communication Networks, Part II*, volume 14974 of *Lecture Notes in Computer Science*, pages 338–359, Amalfi, Italy, September 11–13, 2024. Springer, Cham, Switzerland. doi:10.1007/978-3-031-71073-5_16.

[Des88] Yvo Desmedt. Society and Group Oriented Cryptography: A New Concept. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume

293 of *Lecture Notes in Computer Science*, pages 120–127, Santa Barbara, CA, USA, August 16–20, 1988. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-48184-2_8.

[DKLS18] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Secure two-party threshold ecdsa from ecdsa assumptions. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 980–997, 2018. doi:10.1109/SP.2018.00036.

[DKLS19] J. Doerner, Y. Kondi, E. Lee, and A. Shelat. Threshold ECDSA from ECDSA Assumptions: The Multiparty Case. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1051–1066, 2019. doi:10.1109/SP.2019.00024.

[DKLS24] Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. Threshold ecdsa in three rounds. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3053–3071, 2024. doi:10.1109/SP54263.2024.00178.

[DKLS25] Antonín Dufka, Semjon Kravtšenko, Peeter Laud, and Nikita Snetkov. Trilithium: Efficient and universally composable distributed ML-DSA signing. Cryptology ePrint Archive, Paper 2025/675, 2025. URL: https://eprint.iacr.org/2025/675.

[Eur25] European Commission. The European Digital Identity Wallet Architecture and Reference Framework V2.4.0, 2025. URL: https://eudi.dev/2.4.0/architecture-and-reference-framework-main/.

[Fis05] Marc Fischlin. Communication-efficient non-interactive proofs of knowledge with online extractors. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 152–168, Santa Barbara, CA, USA, August 14–18, 2005. Springer Berlin Heidelberg, Germany. doi:10.1007/11535218_10.

[FS87] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-47721-7_12.

[Gan95] R. Ganesan. Yaksha: augmenting kerberos with public key cryptography. In *Proceedings of the Symposium on Network and Distributed System Security*, pages 132–143, 1995. doi:10.1109/NDSS.1995.390639.

[GCRH23] Alexandre Augusto Giron, Ricardo Custódio, and Francisco Rodríguez-Henríquez. Post-quantum hybrid key exchange: a systematic mapping study. *Journal of Cryptographic Engineering*, 13(1):71–88, Apr 2023. doi:10.1007/s13389-022-00288-9.

[GT12] Kristian Gjøsteen and Øystein Thuen. Password-Based Signatures. In Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul, editors, *Public Key Infrastructures, Services and Applications*, pages 17–33, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-29804-2_2.

[Hin08] M Jason Hinek. On the security of multi-prime RSA. *Journal of Mathematical Cryptology*, 2(2):117–147, 2008. doi:10.1515/JMC.2008.006.

[ISO21] ISO Central Secretary. Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application. Standard ISO/IEC 18013-5:2021, International Organization for Standardization, Geneva, CH, 2021. URL: https://www.iso.org/standard/69084.html.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Secur.*, 1(1):36–63, August 2001. doi:10.1007/s102070100002.

[KOS15] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741, Santa Barbara, CA, USA, August 16–20, 2015. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-662-47989-6_35.

[Lin20] Yehuda Lindell. Secure multiparty computation. *Commun. ACM*, 64(1):86–96, December 2020. doi:10.1145/3387108.

[Lin21] Yehuda Lindell. Fast Secure Two-Party ECDSA Signing. *Journal of Cryptology*, 34(4):44, October 2021. doi:10.1007/s00145-021-09409-9.

[LN18] Yehuda Lindell and Ariel Nof. Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1837–1854, Toronto, ON, Canada, October 15–19, 2018. ACM Press. doi:10.1145/3243734.3243788.

[LR22] Anna Lysyanskaya and Leah Namisa Rosenbloom. Universally Composable Σ-protocols in the Global Random-Oracle Model. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022: 20th Theory of Cryptography Conference, Part I*, volume 13747 of *Lecture Notes in Computer Science*, pages 203–233, Chicago, IL, USA, November 7–10, 2022. Springer, Cham, Switzerland. doi:10.1007/978-3-031-22318-1_8.

[LV22] Peeter Laud and Jelizaveta Vakarjuk. A comparison-based methodology for the security assurance of novel systems. In Sokratis K. Katsikas, Frédéric Cuppens, Christos Kalloniatis, John Mylopoulos, Frank Pallas, Jörg Pohle, M. Angela Sasse, Habtamu Abie, Silvio Ranise, Luca Verderame, Enrico Cambiaso, Jorge Maestre Vidal, Marco Antonio Sotelo Monge, Massimiliano Albanese, Basel Katt, Sandeep Pirbhulal, and Ankur Shukla, editors, *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*, volume 13785 of *Lecture Notes in Computer Science*, pages 625–644. Springer, 2022. doi:10.1007/978-3-031-25460-4_36.

[MR01a] Philip D. MacKenzie and Michael K. Reiter. Delegation of Cryptographic Servers for Capture-Resilient Devices. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001: 8th Conference on Computer and Communications Security*, pages 10–19, Philadelphia, PA, USA, November 5–8, 2001. ACM Press. doi:10.1145/501983.501986.

[MR01b] Philip D. MacKenzie and Michael K. Reiter. Networked Cryptographic Devices Resilient to Capture. In *2001 IEEE Symposium on Security and Privacy*, pages 12–25, Oakland, CA, USA, May 2001. IEEE Computer Society Press. doi:10.1109/SECPRI.2001.924284.

[Oka88] Tatsuaki Okamoto. A digital multisignature scheme using bijective public-key cryptosystems. *ACM Transactions on Computer Systems (TOCS)*, 6(4):432–441, 1988. doi:10.1145/48012.48246.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238, Prague, Czech Republic, May 2–6, 1999. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-48910-X_16.

[Pas03] Rafael Pass. On Deniability in the Common Reference String and Random Oracle Model. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 316–337, Santa Barbara, CA, USA, August 17–21, 2003. Springer Berlin Heidelberg, Germany. doi:10.1007/978-3-540-45146-4_19.

[PW01] Birgit Pfitzmann and Michael Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *2001 IEEE Symposium on Security and Privacy*, pages 184–200, Oakland, CA, USA, May 2001. IEEE Computer Society Press. doi:10.1109/SECPRI.2001.924298.

[Roy22] Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 657–687, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Cham, Switzerland. doi:10.1007/978-3-031-15802-5_23.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978. doi:10.1145/359340.359342.

[Sar19] Augustin 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.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 239–252, Santa Barbara, CA, USA, August 20–24, 1990. Springer, New York, USA. doi:10.1007/0-387-34805-0_22.

[SFG23] Douglas Stebila, Scott Fluhrer, and Shay Gueron. Hybrid key exchange in TLS 1.3. Internet-draft, Internet Engineering Task Force, September 2023. URL: https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/09/.

[SLCH25] Manu Sporny, Dave Longley, David Chadwick, and Ivan Herman. Verifiable Credentials Data Model v2.0, 2025. URL: https://www.w3.org/TR/vc-data-model-2.0/.

[XAX+21] Haiyang Xue, Man Ho Au, Xiang Xie, Tsz Hon Yuen, and Handong Cui. Efficient Online-friendly Two-Party ECDSA Signature. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 558–573, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press. doi:10.1145/3460120.3484803.

[XS03] Shouhuai Xu and Ravi S. Sandhu. Two Efficient and Provably Secure Schemes for Server-Assisted Threshold Signatures. In Marc Joye, editor, *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*,

pages 355–372, San Francisco, CA, USA, April 13–17, 2003. Springer Berlin Heidelberg, Germany. doi:10.1007/3-540-36563-X_25.

A Relation between $\mathcal{F}^{\mathsf{gSpl}}$ and $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$

There exists a protocol Π^{Spl} that securely realizes $\mathcal{F}^{\mathsf{gSpl}}$ (serving n parties) in the $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$-hybrid model, as long as the signature scheme Sig is UF-CMA-secure. This system Π^{Spl}, given in Fig. 16, consists of the machine $\mathcal{M}_{\mathrm{cli}}$, the machine $\mathcal{M}_{\mathrm{srv}}$, $(n-2)$ copies of the machine $\mathcal{M}_{\mathrm{gen}}$, and the functionality $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. We see that except for signature verification, the machines mostly forward the messages between $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ and $\mathcal{Z}$, sometimes changing the format of the message (e.g. during key generation). Note also that the machines $\mathcal{M}$ cannot be corrupted (for some notion of "corruption") by the adversary. Instead, $\mathcal{A}$ can corrupt the client and/or the server by sending the appropriate message to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$.

Protocol Π^{Spl}:
Π^{Spl} offers an interface towards $\mathcal{Z}$ for n users: client, server, and $(n-2)$ entities that only verify signatures.
Π^{Spl} consists of the machine $\mathcal{M}_{\mathrm{cli}}$ serving the user P_1, the machine $\mathcal{M}_{\mathrm{srv}}$ serving the user P_2, $(n-2)$ machines $\mathcal{M}_{\mathrm{gen}}$ serving the users $P_3, \ldots, P_n$, and the machine $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. The machine $\mathcal{M}_{\mathrm{cli}}$ has connections to all machines $\mathcal{M}_{\mathrm{gen}}$. The machine $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ is connected to $\mathcal{M}_{\mathrm{cli}}$ and $\mathcal{M}_{\mathrm{srv}}$.

Key generation:

- When the machine $\mathcal{M}_{\mathrm{cli}}$ gets the input $(\mathsf{keygen}, \mathsf{sid}, L, T_0, \mathsf{PIN})$ from P_1, it forwards that message to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. When $\mathcal{M}_{\mathrm{cli}}$ gets back pk, it sends $(\mathsf{key}, \mathsf{sid}, \mathsf{pk})$ to all machines $\mathcal{M}_{\mathrm{gen}}$. Also, $\mathcal{M}_{\mathrm{cli}}$ constructs the description of the algorithm $\mathsf{Ver}(\cdot,\cdot) := \mathsf{Sig.Verify}(\mathsf{pk},\cdot,\cdot)$ and returns $(\mathsf{key}, \mathsf{sid}, \mathsf{Ver})$ to P_1.
- When the machine $\mathcal{M}_{\mathrm{srv}}$ gets the input $(\mathsf{keygen}, \mathsf{sid}, T_0)$ from P_2, it also forwards that message to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. Similarly to $\mathcal{M}_{\mathrm{cli}}$, it gets back pk, constructs Ver and sends $(\mathsf{key}, \mathsf{sid}, \mathsf{Ver})$ to P_2.
- When any of the machines $\mathcal{M}_{\mathrm{gen}}$ receives $(\mathsf{key}, \mathsf{sid}, \mathsf{pk})$ from $\mathcal{M}_{\mathrm{cli}}$, then it stores pk.

Signing:

- When the machine $\mathcal{M}_{\mathrm{cli}}$ gets the input $(\mathsf{sign}, \mathsf{sid}, M, \mathsf{PIN}')$ from P_1, it forwards that message to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. When it gets back $(\mathsf{sign\text{-}success}, \mathsf{sid}, M, \sigma)$ or $(\mathsf{sign\text{-}fail}, \mathsf{sid})$, it forwards that message back to P_1.
- When the machine $\mathcal{M}_{\mathrm{srv}}$ gets the input $(\mathsf{sign}, \mathsf{sid})$ from P_2, it also forwards that message to $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$. Also, $\mathcal{M}_{\mathrm{srv}}$ forwards the reply from $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ back to P_2.

Signature verification:

- When any of the machines $\mathcal{M}$ receives $(\mathsf{verify}, \mathsf{sid}, M, \sigma)$ from $\mathcal{Z}$ and if it has already stored pk, then it runs $b \leftarrow \mathsf{Ver}(\mathsf{pk}, M, \sigma)$ and returns b.

Corruptions:

- When $\mathcal{M}_{\mathrm{cli}}$ or $\mathcal{M}_{\mathrm{srv}}$ is notified by $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$ of its increased corruption level, then it forwards that notification to $\mathcal{Z}$ (either P_1 or P_2).

Figure 16: Protocol Π^{Spl} securely implementing $\mathcal{F}^{\mathsf{gSpl}}$ in $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$-hybrid model

Theorem 9. *If the signature scheme Sig is UF-CMA secure, then protocol Π^{Spl} securely implements $\mathcal{F}^{\mathsf{gSpl}}$ in $\mathcal{F}^{\mathsf{Spl}}_{\mathsf{Sig}}$-hybrid model.*

Proof. We construct a simulator *Sim* that translates between the messages from $\mathcal{F}^{\mathsf{gSpl}}$ and the messages from $\mathcal{A}$. Internally, *Sim* simulates $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$. Next, we describe the way how the simulator *Sim* responds to the commands:

Key generation: This is initiated by the environment, where P_1 and P_2 send the requests to $\mathcal{M}_{\text{cli}}$ and $\mathcal{M}_{\text{srv}}$, but these requests are received by $\mathcal{F}^{\mathsf{gSpl}}$ instead. The functionality $\mathcal{F}^{\mathsf{gSpl}}$ eventually makes the query $(\mathsf{keygen\text{-}init}, \mathsf{sid})$ to the adversary, this query is received by the simulator. It simulates the working of $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$ during key generation (ignoring the values PIN, T_0, L), including the communications with $\mathcal{A}$. During this, *Sim* generates $(\mathsf{pk}, \mathsf{sk})$ (even if either client or server is corrupt). Sends $(\mathsf{key}, \mathsf{Ver}(\mathsf{pk}, \cdot, \cdot))$ to $\mathcal{F}^{\mathsf{gSpl}}$.

Signing:

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

Corruptions

The adversary $\mathcal{A}$ makes corruption requests to $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$, i.e. they reach *Sim*. The simulator forwards them to $\mathcal{F}^{\mathsf{gSpl}}$. If $\mathcal{F}^{\mathsf{gSpl}}$ sends back PIN, then *Sim* forwards it to $\mathcal{A}$. If, according to the internal logic of $\mathcal{F}_{\mathsf{Sig}}^{\mathsf{Spl}}$, 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 $\mathcal{F}^{\mathsf{gSpl}}$. The functionality $\mathcal{F}^{\mathsf{gSpl}}$ may decide on the result by itself, or it may forward the verification request to the simulator. The simulator then answers with $\mathsf{Ver}(\mathsf{pk}, M, \sigma)$.

Security of verification follows from the UF-CMA security of Sig. As long as one of client and server is uncorrupted, the adversary does not have access to sk. Hence if $\mathcal{M}_{\text{gen}}$ would return that a signature verifies, but there is no (M, σ, b) recorded in the database of $\mathcal{F}^{\mathsf{gSpl}}$, 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 $(\mathsf{pk}, \mathsf{sk})$ 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.

□