UC4Free!
Existing Threshold Signatures are UC Secure

Jan Bobolz[1], Elizabeth Crites[2], Markulf Kohlweiss[1,3], Akira Takahashi[4]

[1] University of Edinburgh, UK
[2] Parity Technologies, UK
[3] Input | Output, UK
[4] JPMorgan AI Research & AlgoCRYPT CoE, US

May 9, 2026

Abstract. Threshold signatures have received considerable attention in recent years due to ongoing standardization efforts and deployment in real-world systems. In this work, we prove the universal composability of a wide range of threshold signature schemes, including state-of-the-art protocols compatible with standard signatures used in practice, such as BLS and Schnorr signatures, as well as emerging post-quantum solutions. Importantly, we show UC security without any modifications to the existing protocols.

To this end, we design natural game-based definitions to capture different combinations of main threshold signature scheme properties, such as different levels of unforgeability, adaptive corruption, robustness, and different degrees of preprocessing. These definitions generalize prior definitional work, such as Bellare et al. (CRYPTO'22), and cover a wide range of existing schemes. Moreover, we identify and resolve gaps in prior work. We then express these properties in terms of a UC ideal functionality $\mathcal{F}$-TS3. We prove that a threshold signature scheme UC-realizes $\mathcal{F}$-TS3 if and only if it satisfies our game-based definitions.

This opens up the usage of (existing) threshold signature schemes in a UC setting, enabling scheme designers to formulate their protocols relative to an ideal threshold signature functionality and use the UC composition theorem to argue security given any concrete instantiation. To further support UC scheme designers and to give further guidance on UC modeling for threshold signatures, we provide additional ideal threshold signature functionalities $\mathcal{F}$-TS2, $\mathcal{F}$-TS1, $\mathcal{F}$-TSSync2, and $\mathcal{F}$-TSSync1, which capture fewer properties than $\mathcal{F}$-TS3 but are more convenient to use. $\mathcal{F}$-TS2, $\mathcal{F}$-TS1, $\mathcal{F}$-TSSync2, $\mathcal{F}$-TSSync1 can also be UC-realized by schemes proven secure according to our game-based definitions.

Through this work, we show that composable security does not require sacrificing performance, but it does require rigor when setting up game-based definitions and ideal functionalities.

jan.bobolz@ed.ac.uk
elizabeth@parity.io
markulf.kohlweiss@ed.ac.uk
takahashi.akira.58s@gmail.com

Table of Contents

1 Introduction . . . 3
2 Preliminaries . . . 7
2.1 Universal Composability . . . 9
3 Game-based Definitions for Threshold Signatures . . . 11
3.1 Syntax of Threshold Signature Schemes . . . 11
3.2 Correctness . . . 12
3.3 Robustness . . . 14
3.4 Unforgeability . . . 16
3.5 The interplay between correctness, robustness, and unforgeability . . . 18
4 $\mathcal{F}$-TS3: Fine-grained Threshold Signature Functionality . . . 19
5 $\mathcal{F}$-TS2: Moderate Threshold Functionality . . . 23
6 $\mathcal{F}$-TS1: Simplest Threshold Functionality . . . 27
7 A Moderate UC Functionality for the Synchronous Setting . . . 28
8 A Simple UC Functionality for the Synchronous Setting . . . 32
9 Adapting Our Results to Idealized Models . . . 32
A Related work . . . 38
A.1 Comparison with other Game-based Hierarchies for Threshold Signatures . . . 38
A.2 Comparison with other UC Functionalities for Threshold Signatures . . . 38
B Warm-up: Digital Signature Functionality . . . 39
C A Remark on SUF-0 and SUF-1 . . . 40
D Proof of Theorem 2 (Π-TS3 UC-realizes $\mathcal{F}$-TS3) . . . 41
E Proof of Theorem 3 (Π-TS2 UC-realizes $\mathcal{F}$-TS2) . . . 45
F Proof of Theorem 4 (Π-TS1 UC-realizes $\mathcal{F}$-TS1) . . . 48
G Π-TSSync2 and Proof of Theorem 5 (Π-TSSync2 UC-realizes $\mathcal{F}$-TSSync2) . . . 49
H $\mathcal{F}$-TSSync1 and Π-TSSync1 . . . 52
I Example Instantiations . . . 52
I.1 FROST1 . . . 52
I.2 Threshold Raccoon . . . 54
J Glossary of boxed items . . . 57

1 Introduction

Since their inception in the 1980s [Des88, DF90], threshold signatures have become a critical building block in complex, high-value applications, such as cryptocurrency wallets, decentralized control over smart contracts, digital asset and identity management, and certificate infrastructures. [AS20, AHS20]

Threshold signatures are an attractive option for deployment in such systems due to their failure resilience properties. A single signing key associated with a digital wallet or identity credential represents a single point of failure and potential attack. Threshold signatures allow for the distribution of a (virtual) signing key among a set of parties such that any quorum above the threshold can collectively produce a valid signature, but any smaller subset cannot. Importantly, threshold signatures can tolerate some fraction of adversarial participants and remain secure against forgeries. Moreover, they increase the availability of the respective signing service.

Motivated by their widespread use in practice, the US National Institute for Standards and Technology (NIST) recently announced the First Call for Multi-Party Threshold Schemes [BP25]. The Call outlines a range of corruption profiles and security properties of interest. For example, there is a strong preference for schemes with provable adaptive security, a strong notion of security against corruption of parties chosen adaptively as the protocol progresses. The Call also emphasizes security analyses under various forms of composition, including concurrent executions of the protocol with itself, and, ideally, with other protocols as part of a larger system. Threshold signature schemes are deployed in complex systems, such as blockchains, which are typically proven in the UC setting.

Many modern threshold signature schemes are proven secure in the game-based setting, where an idealized game defines the goals and capabilities of an adversary. Many others define security in the form of a secure-by-definition ideal functionality, where protocols are secure if attacks against them can be translated to attacks against the ideal functionality. To strike a balance between strong security properties and compositional guarantees, the Call suggests security proofs in *both* settings – for example, UC analysis of static security in combination with game-based analysis of adaptive security. (Re)proving security in multiple settings introduces prohibitive overhead for protocol designers, and creates a dissatisfactory theoretical gap and piecemeal approach to security. In light of this, we consider the following question:

What compositional guarantees do existing threshold signature schemes satisfy?

Our work has a simple and positive message. We show that many threshold signature schemes, and in particular the game-based notions under which they are proven secure, are stronger than advertised. Namely, they satisfy natural and strong compositional guarantees. Assuming their game-based security proofs under strong, concurrent models, such as those of Bellare et al. [BCK+22, BTZ22], are correct (and not, for example, insecure due to attacks [BLL+22, CS25]), they are also universally composable in the traditional cryptographic sense. Thus, our results show that many schemes can be safely deployed as a subcomponent of large, complex systems with secure composability with other subprotocols. This is the threshold signature counterpart to the well-known result that game-based (EUF-CMA) security implies UC security for single-party signatures [Can04].

Refining game-based security notions. As a first step towards analyzing the composability of game-based threshold signature schemes, we first refine, generalize, and unify existing game-based security notions.

Like several existing works, we define several levels of unforgeability of increasing strength: UF-0 to UF-4, and SUF-2 to SUF-4, where SUF denotes strong unforgeability. In the standard unforgeability game for single-party signatures [GMR88], the adversary is given access to a signing oracle and outputs a forgery, which is valid if it is non-trivial. In this context, a forgery is non-trivial if the adversary never queried its signing oracle on the forgery message. Strong unforgeability ensures that an adversary cannot produce a new signature on a message for which it has already seen a valid signature. In this case, the forgery is considered non-trivial if the adversary never received a signature on its forgery message. Defining non-triviality conditions for the forgery in the threshold setting introduces unique challenges, first explored in the work of Bellare et al. [BCK+22, BTZ22].

Our game-based security hierarchy implies the hierarchy of Bellare et al., and thus their findings, including strong unforgeability, translate to our setting. However, their hierarchy only applies to partially non-interactive schemes, which consist of one online signing round and at most one offline, preprocessing round, before the message and signing set are known – the canonical examples being

FROST [KG20, BCK+22] and threshold BLS [Bol03]. Somewhat surprisingly, their hierarchy cannot identify threshold BLS as strongly unforgeable and only establishes UF-1 security, even though it has unique signatures per message. We fill this gap by proving that threshold BLS achieves SUF-2 security in our model.

Cremers et al. [CPZ24] extend the definitions of Bellare et al. to cover interactive signing and key generation phases as well as adaptive corruption in both phases. This is a valuable expansion of the game-based landscape. It does, however, fall short in one important aspect: it assumes that no scheme without preprocessing can be strongly unforgeable. Strong unforgeability is a critical security feature in many applications of signatures. Indeed, the malleability of ECDSA signatures is a well-known vulnerability that allows for two valid signatures for the same message. In the blockchain setting, this so-called transaction malleability allows an adversary to change the identifier for a transaction, thus opening the door to duplicate transactions. This has been exploited in high-profile attacks [DW14, ADMM15] and attracted significant attention in recent years [Wui14, LW16, WNR20]. Given such attacks, the NIST Call specifies strong unforgeability as a required property of any candidate submission. Our security notions fill this gap, capturing strong unforgeability for interactive schemes with preprocessing. Moreover, we show that the SUF-0 and SUF-1 notions in [CPZ24] capture a weak form of strong unforgeability, allowing for highly malleable signatures.

Threshold signature schemes (assuming idealized key generation)[5] covered by these hierarchies are also covered by our hierarchy. (A more comprehensive list of prior game-based hierarchies is given in Appendix A.1.) Interestingly, the property of robustness is not considered in any prior hierarchy. Robustness, or guaranteed output delivery (G.O.D.), is an important feature in consensus-level protocols (i.e., systems involving hundreds or thousands of parties), which ensures a valid signature can always be generated, up to some bound on the number of corrupted parties. The NIST Call specifies various numbers of parties and corruption profiles of interest, including schemes secure for up to 1024 parties. Inspired by the first game-based definition proposed by Ruffing et al. [RRJ+22], we provide a robustness definition as a natural extension of correctness/unforgeability games, where the adversary can entirely control scheduling of protocol messages, and wins if it can prevent honest parties from producing a valid signature.

Taken together, our definitional framework gives a strong foundation for protocol designers, capturing a wide array of security goals defined as intuitive properties that can be be hand-picked to specification. Our results impact a wide range of existing threshold signature schemes proven secure in the game-based setting, including state-of-the-art protocols compatible with standard signatures used in practice, such as BLS and Schnorr signatures, as well as emerging post-quantum solutions (Table 1). Standardization efforts for some protocols, such as FROST [CKGW22], are already underway.

Composable security. Game-based definitions model properties as a game that an adversary tries to win. Put simply, (non-threshold) signature unforgeability is defined in terms of a game that allows an adversary $\mathcal{A}$ to ask for signatures from an honest signer and ends with $\mathcal{A}$ outputting a signature σ^* on a message m^*. If σ^* is a valid signature and $\mathcal{A}$ has not asked for a signature on m^*, then $\mathcal{A}$ *wins* the game. Security is the absence of adversaries that win the game with non-negligible probability.

Universally composable security (UC [Can00, Can01, CDPW07, Can20, BCH+20]) instead defines security in terms of an *ideal functionality*, which models a scheme that is *secure by definition*. Very informally, a signature (ideal) functionality $\mathcal{F}$-Sig [Can04], see Functionality 12, exposes an interface $\sigma \leftarrow \textsc{Sign}(m)$ and an interface $\{0,1\} \leftarrow \textsc{Verify}(\mathsf{pk}, m, \sigma)$, modeling the usual syntax of a signature scheme (minus secret keys). However, the ideal functionality keeps track of all messages m for which it created signatures. If $\textsc{Verify}(\mathsf{pk}, m, \sigma)$ is queried on a message m for which no signature has been created, then it is programmed to output 0 and declare the signature invalid no matter what. This is how unforgeability is captured in the ideal functionality: Forgeries *do not exist* because signatures for unqueried messages are simply rejected by Verify.

So what does it mean for a signature scheme to be UC secure? The signature scheme UC-protocol Π-Sig exposes the same interfaces Sign, Verify as the ideal functionality $\mathcal{F}$-Sig. However, instead of keeping a list of messages and signatures in memory, it would simply run its Sign or Verify algorithm when its Sign or Verify interface is invoked. Very informally, we say that Π-Sig UC-realizes $\mathcal{F}$-Sig (is "secure") if no environment can distinguish whether it interacts with Π-Sig (where forgeries are

[5] Unforgeability is often proved with respect to trusted key generation, which cleanly separates the assumptions for secure signing vs. secure key generation, cf. Table 1. See Section 4 for more details.

Table 1: Summary of existing pre-quantum (top) and post-quantum (bottom) threshold signature schemes. This list is not exhaustive. All schemes are secure in the random oracle model (ROM). We do not cover security in the algebraic group model (AGM); see Section 9. '(S)UF-X' denotes game-based (strong) unforgeability as defined in Game 3. '1/2' denotes that the scheme is adaptively secure for up to $(t_{\text{correct}} - 1)/2$ corrupted parties (versus a full $t_{\text{correct}} - 1$), where t_{correct} is the threshold. '$\mathsf{rnd}_{\text{sign}}$' denotes the total number of signing rounds; '$\mathsf{rnd}_{\text{com}}$' denotes the committing round in which all the necessary inputs are committed (e.g., the signing set and message). We specify more fine-grained preprocessing in Section 3.1. All schemes are proven secure with idealized key generation. The assumptions are: DL = discrete logarithm; CDL = circular DL; CDH = computational Diffie-Hellman; DDH = decisional Diffie-Hellman; OMDL = one-more discrete logarithm; AOMDL = algebraic OMDL; AOMCDH = algebraic one-more CDH; VCDH = vector CDH; MSIS = module short integer solution; MLWE = module learning with errors; AOM-MLWE = algebraic one-more MLWE; ML-DSA = security of ML-DSA signatures.

Scheme	UF Level	Adaptive	Assumptions	$\mathsf{rnd}_{\text{sign}}$	$\mathsf{rnd}_{\text{com}}$
Threshold BLS_{I} [Bol03]	UF-0		GDH	1	1
[BTZ22, BCK+22]	UF-1		t_{correct}-VCDH		
Threshold $\text{BLS}_{\text{II,III}}$ [DR24]	UF-0	✓	DDH, Co-CDH	1	1
FROST [KG20]	–		–	2	2
[BTZ22, BCK+22]	SUF-3		OMDL		
[CKK+25]	UF-0	✓ (1/2)	AOMDL		
FROST2 [CKM21]	UF-0		OMDL	2	2
[BTZ22, BCK+22]	SUF-2		OMDL		
[CKK+25]	UF-0	✓ (1/2)	AOMDL		
FROST3 [RRJ+22, CGRS23]	UF-0		OMDL	2	2
[CKK+25]	UF-0	✓ (1/2)	AOMDL		
FROST-Mask [Che25]	UF-0	✓	AOMDL	2	2
Sparkle+ [CKM23]	UF-0		DL	3	2
[CKM23]	UF-0	✓ (1/2)	AOMDL		
[CFOS25]	UF-0		CDL		
Twinkle [BLT+24]	UF-0	✓	AOMCDH	3	1
[BLT+24]	UF-0	✓	DDH	3	1
Gargos [BDLR25a]	UF-0	✓	DDH	3	1
Snap [KRT24]	UF-1	✓	DL	4	2
Crackle [KRT24]	UF-1	✓	DL	5	3
Glacius [BDLR25b]	UF-0	✓	DL	5	1
Finally [BCdP+25]	UF-0		MLWE, MSIS	3	1
Threshold ML-DSA [CdPE+26]	UF-0		ML-DSA, MLWE	3	1
T-Raccoon [DKM+24]	UF-0		MLWE, MSIS	3	1
T-Raccoon-KRT [KRT24]	UF-1		MLWE, MSIS	3	1
T-Raccoon-KRT-4-rnd	UF-1	✓	MLWE, MSIS	4	1
T-Raccoon-KRT-5-rnd	UF-1	✓	MLWE, MSIS	5	2
T-Raccoon-IA [PKN+25]	UF-1		MLWE, MSIS	3	1

hopefully hard to find but would in principle be accepted by VERIFY) or with $\mathcal{F}$-Sig (where forgeries would be rejected by VERIFY). While this explanation is highly informal and simplified, it illustrates the main difference between the paradigms of game-based security and UC security. (We give an introduction to UC security in Section 2.1.)

The main advantage of this modeling is composability: When writing a larger protocol, one can easily use $\mathcal{F}$-Sig as a subprotocol. Due to the absence of forgeries in $\mathcal{F}$-Sig by definition, this makes the analysis of the larger protocol rather straightforward. Then, because UC-emulation means that Π-Sig and $\mathcal{F}$-Sig are indistinguishable, one can replace the $\mathcal{F}$-Sig subprotocol with Π-Sig. This intuition is formalized in the UC composition theorem [Can20, BCH+20]. As a result, one can show that the larger protocol is still secure when using the real signature scheme.

Our threshold signature functionalities. Designing ideal functionalities is not always straightforward, especially for complex scheme types such as threshold signatures. There is a tension between abstraction (hiding unnecessary details and making the functionality easy to understand and use), and guarantees encoded in the functionality. The more abstract and simple the functionality, the more difficult it is to encode desired security guarantees. For example, UC makes it easy for a functionality to abstract away from the number of rounds a signing process takes, but this also means that functionality cannot guarantee any properties about the number of rounds. In order to cater to different use cases, and to give guidance on designing different threshold signature functionalities, we introduce several ideal threshold signature functionalities in this paper, with varying trade-offs between abstraction and guarantees. Our ideal threshold signature functionalities are as follows.
$\mathcal{F}$-TS3 exposes the threshold signature scheme's round functions and is the most complex of the five functionalities we define. While perhaps cumbersome to use (the interface user needs to take care

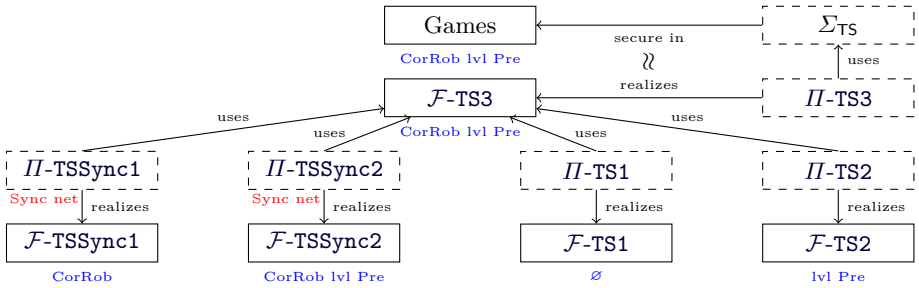

Fig. 1: Overview of our functionalities. Definitions have solid lines, protocols have dashed lines. CorRob = models correctness and robustness; lvl = models all unforgeability levels (S)UF-X; Pre = models preprocessing. Sync net = requires synchronous authenticated network.

of protocol message passing), it allows for very granular bookkeeping and hence is able to capture nuanced correctness, unforgeability, and robustness properties. We prove that $\mathcal{F}$-TS3 is equivalent to game-based definitions in Theorem 2.

$\mathcal{F}$-TS2 abstracts from the concrete round structure, modeling only a separation between a non-interactive (optional) preprocessing phase and an interactive signing phase. This is easier to use (the interface user only needs to pass around preprocessing data alongside signing requests, not the protocol messages), but because this definition leaves the concrete protocol execution up to the ideal UC adversary, $\mathcal{F}$-TS2 cannot guarantee any meaningful notion of correctness or robustness. Hence $\mathcal{F}$-TS2 only guarantees unforgeability. Correctness and, if needed, robustness would need to be captured outside of the UC functionality (e.g., by proving additonal properties of a protocol in addition to UC emulation).

$\mathcal{F}$-TS1 further abstracts away preprocessing and limits itself to capturing the UF-1 unforgeability level (easily adapted to UF-0, see Section 6). This significantly simplifies the functionality, making $\mathcal{F}$-TS1 the easiest one to understand and use, while still covering important properties and protocols, cf. Table 1.

$\mathcal{F}$-TSSync2 and $\mathcal{F}$-TSSync1 are synchronous variants of $\mathcal{F}$-TS2 and $\mathcal{F}$-TS1. They work in a synchronous round model, allowing them to model correctness and robustness, not guaranteed by $\mathcal{F}$-TS2, $\mathcal{F}$-TS1. The trade-off is that $\mathcal{F}$-TSSync2 and $\mathcal{F}$-TSSync1 effectively require a synchronous network to realize.

Figure 1 presents an overview of these functionalities and their relations. The equivalence between $\mathcal{F}$-TS3 and game-based definitions establishes two things: First, because game-based security implies $\mathcal{F}$-TS3 security, a wide range of existing game-based schemes are UC secure and hence safe to compose. Second, because $\mathcal{F}$-TS3 security implies game-based security, $\mathcal{F}$-TS3 must be strong enough to capture all properties modeled by the game-based definitions, giving us confidence in the model. By repeated application of the UC composition theorem, one can effectively implement all of our functionalities using a game-based scheme that is secure at the appropriate level (e.g., $\mathcal{F}$-TS1 can be realized using $\mathcal{F}$-TS3, which can be realized from an appropriate game-based scheme Σ_{TS}). Of course, we are not the first to define ideal functionalities for threshold signatures. We survey existing definitions in Appendix A.2.

On simulation-based security versus UC-security. Oftentimes, when cryptographers mention UC security in the context of threshold signatures, they are referring to a very strong *simulation-based* notion of security. These definitions relate to some underlying (non-threshold) signature scheme, such as ECDSA [Lin17, DKLs18, DKLs19, GS22, DKLs24], Schnorr [Lin24], and ML-DSA [BdCE+25] signatures. Essentially, security requires the existence of a powerful simulator that is able to, given any target signature σ on m from the underlying signature scheme, simulate a protocol execution that produces exactly σ. We refer to this as the $\mathcal{F}$-Sign approach (named after the UC functionality that computes the Sign function of some underlying signature scheme). From a theory standpoint, this is a very satisfying and clean notion [Lin24]: The adversary does not learn anything more than a single valid signature, and they cannot influence the distribution of this signature (except negligibly). Unforgeability is inherited from the underlying signature scheme: the simulator ensures that the multi-party signing protocol does not behave any differently than the underlying signature scheme's signing procedure. One may summarize the unforgeability argument of the resulting threshold signature

scheme as "unforgeable underlying scheme + $\mathcal{F}$-Sign guarantees ⇒ unforgeable threshold signature scheme."

Instead of splitting the unforgeability argument into two parts, the game-based and UC definitions in our paper *directly* define unforgeability for threshold signature schemes. We call this approach the $\mathcal{F}$-tSig approach [CDL+25, Mak22, CMP20, CGG+20, AF04a, ADN06, KKL21] (named after the UC functionality that models an unforgeable threshold signature scheme, which we show to be implied by game-based definitions). This comes with increased flexibility. For example, a game-based threshold signature scheme may allow the adversary to influence the distribution of the output signature in some benign way [Sho25] as long as this influence does not lead to a forgery. This is not allowed in the strong $\mathcal{F}$-Sign definitions, which do not tolerate such deviations from the output distribution.

Ultimately, the lack of such flexibility in $\mathcal{F}$-Sign-style definitions leads to performance trade-offs: Threshold signing protocols following the $\mathcal{F}$-Sign definitional paradigm need to ensure additional properties that may not be strictly necessary for unforgeability. For example, this may involve using straightline extractable proofs of knowledge (e.g., via Fischlin's transform), which the simulator can efficiently extract, instead of Fiat-Shamir based proofs, which are more efficient but cannot be applied in the $\mathcal{F}$-Sign setting due to issues with rewinding. Such rewinding issues can often be circumvented in direct unforgeability proofs in the game-based/$\mathcal{F}$-tSig setting.

While there are good reasons to desire strong $\mathcal{F}$-Sign security (as argued in [Lin24]), an important message of our paper is that existing highly efficient game-based threshold signature schemes (Table 1) are already UC secure without modification by following the perfectly reasonable $\mathcal{F}$-tSig definitional style (Sections 4 to 8) instead of the $\mathcal{F}$-Sign style. Additionally, we note that when it comes to specifying desired security properties of threshold signature schemes, our $\mathcal{F}$-tSig approach has certain advantages over the $\mathcal{F}$-Sign approach. Namely, $\mathcal{F}$-Sign only expresses part of the picture (the signing procedure), while $\mathcal{F}$-tSig captures the full threshold signature scheme. Furthermore, unforgeability guarantees are implicit in $\mathcal{F}$-Sign (and need to be worked out depending on the underlying signature scheme), while $\mathcal{F}$-tSig directly and generically expresses the exact unforgeability guarantees. This makes $\mathcal{F}$-tSig definitions easier to use when designing larger protocols that depend on threshold signature subprotocols.

Paper outline. We begin by specifying basic notation and providing an overview of the universal composability framework in Section 2. We present our hierarchy of game-based definitions for threshold signatures in Section 3, including definitions for correctness, robustness, and unforgeability. We then introduce our most fine-grained threshold signature functionality $\mathcal{F}$-TS3 in Section 4, a moderate functionality $\mathcal{F}$-TS2 in Section 5, and a simple functionality $\mathcal{F}$-TS1 in Section 6. Their synchronous counterparts, $\mathcal{F}$-TSSync2 and $\mathcal{F}$-TSSync1, are given in Section 7 and Section 8, respectively. We discuss idealized models, such as the random oracle model, in Section 9. Full proofs of our results (Theorems 2 to 6) are given in Appendices D to H. We survey related work on game-based threshold signature definitions and UC threshold signature functionalities in Appendix A.

2 Preliminaries

General notation. We use the following notation throughout the paper.

Sets and vectors. We write $[a, b] = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ and $[b] = [1, b]$. For a set S, we write $x \leftarrow_{\text{first}} S$ to retrieve the lexicographically first element of S, and $x \xleftarrow{\$} S$ to retrieve a uniformly random element of S. For a set S, we use generalized vector notation $(a_i)_{i \in S}$ to denote the map $a : S \to \{0,1\}^*$ with $a(i) = a_i$. We can imagine the format for a to be $a = \{(i, a_i) | i \in S\}$. Note that for $S = [n]$, this is compatible with the usual vector modeling, i.e., $(a_i)_{i \in [n]} = (a_1, \ldots, a_n)$. As a consequence, if $(a_i)_{i \in S}$ is given as input to an algorithm, the algorithm learns the set of all a_i as well as the set S and the association of which a_i belongs to which indices $i \in S$.

Threshold signature shorthands. The vector $\mathbf{t} = (t_{\text{correct}}, t_{\text{corrupt}}, t_{\text{UF}}, t_{\text{robust}})$ denotes different thresholds and $\mathbf{rnd} = (\mathsf{rnd}_{\text{T}}, \mathsf{rnd}_{\text{msg}}, \mathsf{rnd}_{\text{ctx}}, \mathsf{rnd}_{\text{com}}, \mathsf{rnd}_{\text{sign}})$ denotes different round numbers, as described in Section 3.1. Table 2 gives an overview of variables in this paper.

Algorithms and pseudocode. ppt denotes probabilistic polynomial time. All algorithms Alg are probabilistic and stateless (unless state is explicit input). UC procedures Proc are probabilistic and have access to state. We use the notation Obj::X to refer to a procedure (or value) X located in the machine/object Obj. The set $\mathbf{H}$ is the set of honest parties (it starts with the full set of parties $\mathbf{H} := \mathbf{P}$); $\mathbf{C}$ is its complement, i.e., the set of corrupted parties (initially, $\mathbf{C} := \varnothing$). The pseudocode

Table 2: Overview of variables and notation.

Notation	Meaning
	Signing process
$\boldsymbol{T}$	Set of parties involved in a specific signing process. $\boldsymbol{T} \subseteq \mathbf{P}$. Generally $\lvert\boldsymbol{T}\rvert \geq t_{\text{correct}}$.
m	Message to be signed.
ctx	Arbitrary context string signers must agree on (input to ssid).
$ssid$	Signing session identifier. Signers must agree on $ssid$. Generated from $\mathsf{ssid}(\dots)$.
$\mathsf{eid}_{\mathcal{P}}$	Party-local signing session identifier. Parties have different $\mathsf{eid}_{\mathcal{P}}$ for the same process.
$\mathsf{pm}_{\mathcal{P},k}$	Protocol message sent by $\mathcal{P}$ in round k.
$\mathsf{pp}_{\mathcal{P}}$	Preprocessing message from $\mathcal{P}$. Alias of $\mathsf{pm}_{\mathcal{P},1}$.
$\mathbf{pm}_k$	Collection of messages from round k ($\mathbf{pm}_k = (\mathsf{pm}_{\mathcal{P},k})_{\mathcal{P}\in\boldsymbol{T}}$).
inp	Inputs to next-round signing oracle. $\mathsf{inp} = (\mathsf{inp}.\mathrm{T}, \mathsf{inp}.\mathrm{m}, \mathsf{inp}.\mathrm{ctx}, \mathsf{inp}.\mathbf{pm})$. Some values are only present in certain rounds. See Oracle 1.
par	Trusted public parameters (one for many groups/keys).
pk	Public verification key.
$\mathsf{sk}_{\mathcal{P}}$	Secret signing key share of party $\mathcal{P}$.
$\mathsf{st}_{\mathcal{P}}$	Global state of $\mathcal{P}$ (across all signing processes).
$\mathsf{st}_{\mathcal{P},\mathsf{eid}}$	Local state of $\mathcal{P}$ for a specific signing process identified by eid.
	Protocol parameters
κ	Security parameter.
$\mathbf{P}$	Set of parties with access to a signing key share.
lvl	Unforgeability level (see Table 3).
ssid	Algorithm to generate $ssid$ (referenced by unforgeability levels).
cor	Corruption style (static or adaptive).
$\mathsf{rnd}_{\mathrm{T}}$	Round where $\boldsymbol{T}$ must be input ($\mathsf{rnd}_{\mathrm{T}} > 1$ models preprocessing).
$\mathsf{rnd}_{\mathrm{msg}}$	Round where m must be input ($\mathsf{rnd}_{\mathrm{msg}} > 1$ models preprocessing).
$\mathsf{rnd}_{\mathrm{ctx}}$	Round where ctx must be input ($\mathsf{rnd}_{\mathrm{ctx}} > 1$ models preprocessing).
$\mathsf{rnd}_{\mathrm{com}}$	Round after which a message is considered "signed" w.r.t. unforgeability.
$\mathsf{rnd}_{\mathrm{sign}}$	Total number of rounds for the signing protocol (including preprocessing).
t_{correct}	Correctness guarantees require $\lvert\boldsymbol{T}\rvert \geq t_{\text{correct}}$.
t_{robust}	Robustness guarantees require $\lvert\boldsymbol{T}\rvert \geq t_{\text{robust}}$.
t_{corrupt}	Unforgeability and robustness are void if $\lvert\mathbf{C}\rvert \geq t_{\text{corrupt}}$. (We require $t_{\text{robust}} \leq t_{\text{corrupt}}$.)
t_{UF}	If t_{UF} signers (honest and corrupt) agree to sign, then forgery becomes trivial. (We require $1 \leq t_{\text{UF}} \leq t_{\text{correct}}, t_{\text{robust}}$.)
	UC
$\mathcal{Z}$	Environment.
$\mathcal{A}$	Adversary in the real world.
$\mathcal{S}$	Adversary in the ideal world (simulator).
$\mathcal{P}$	Party identifier (often called pid). Same as in game-based definition.
sid	UC session identifier.
$\clockwise$	Indicator that call yields control to the adversary (may not return immediately).
$(\clockwise)$	Indicator that procedure calls the adversary (directly or through subroutine).
$\mathbf{C}$	Set of corrupted parties.
$\mathbf{H}$	Set of honest parties.

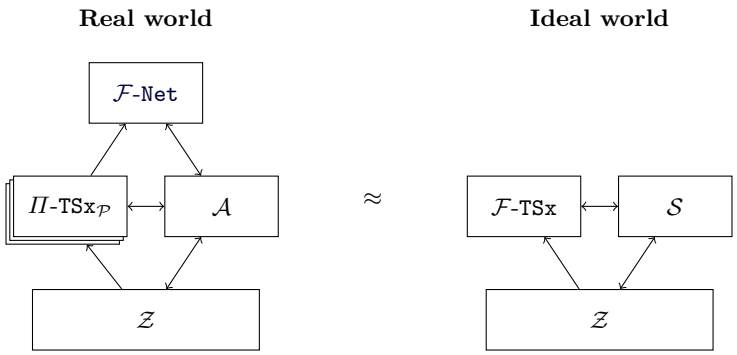

Fig. 2: An illustration of our universal composability setting. We omit the dummy parties for $\mathcal{F}$-TSx.

statement "**assert** ϕ" translates to "if $\neg\phi$ then **return** $\perp$" (i.e., the current procedure stops executing and returns the error symbol). In UC pseudocode, we use the symbol $\clubsuit$ when calling UC adversary (/simulator) to emphasize that this call may not immediately return: the call yields control to the adversary, who may yield control to the environment, which may start new (concurrent) processes before returning. The symbol ($\clubsuit$) in front of a procedure indicates that it calls the UC adversary (directly or through one of its subroutines), effectively also meaning that it may not return immediately. The $\clubsuit$ and ($\clubsuit$) symbols do not modify a call or procedure, they are purely informational.

2.1 Universal Composability

The Universal Composability (UC) framework [Can01, CDPW07, Can00, BCH+20, Can20] establishes composable security of a cryptographic protocol. We summarize a version of the UC framework from [Can00] that supports global subroutines [BCH+20]. In contrast with conventional UC, and in accordance with established practice for game-based security modeling [BR06], we consider uniform environments (i.e., the environment takes the security parameter κ as input instead of the environment being *chosen dependending on* κ). Our results carry over to a non-uniform UC setting if non-uniformity is added to all game-based definitions, assumptions, and proofs. While this would potentially improve the modeling of pre-computation based attacks [BL13], the uniform modeling has the advantage of linking directly to existing game-based results, as well as being simpler.

Functionality 1: $\mathcal{F}$-Net

An insecure point-to-point communication channel with no delivery guarantees.

($\clubsuit$)$\mathcal{P}$.Send($\mathcal{Q}$, *port*, pm) for all $\mathcal{P}$ // Party $\mathcal{P}$ sends message pm to $\mathcal{Q}$ on port *port*
1: $\clubsuit\mathcal{A}$::Send($\mathcal{P}$, $\mathcal{Q}$, *port*, pm) // Adversary sees messages we send immediately upon sending.

($\clubsuit$)$\mathcal{P}$.Receive($\mathcal{Q}$, *port*) for all $\mathcal{P}$ // Party $\mathcal{P}$ checks for next message from $\mathcal{Q}$ on port *port*.
2: **return** pm $\leftarrow$ $\clubsuit\mathcal{A}$::Receive($\mathcal{P}$, $\mathcal{Q}$, *port*) // Adversary decides what (if any) message is received, and when.

Functionality 2: $\mathcal{G}$-Setup[Setup]

$\mathcal{G}$-Setup is a (global) functionality parameterized by the algorithm Setup.

Init() // Invoked once upon creation
1: par $\leftarrow$ Setup(1^κ)

$\mathcal{P}$.Setup() for all $\mathcal{P}$
2: **return** par

Functionality 3: $\mathcal{F}$-Gen[$\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}$::Setup, Σ_{TS}::KeyGen]

with access to $\mathcal{G}$-Setup[Σ_{TS}::Setup].

Init() // Invoked once upon creation

1: par $\leftarrow$ $\mathcal{G}$-Setup[Setup]::$\mathcal{P}$.Setup()

2: $(\mathsf{pk}, \{\mathsf{sk}_{\mathcal{Q}}\}_{\mathcal{Q}\in\mathbf{P}}) \leftarrow \mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})$

$\mathcal{P}$.KeyGen() for $\mathcal{P} \in \mathbf{P}$

3: **return** $(\mathsf{pk}, \mathsf{sk}_{\mathcal{P}})$

Overview of the UC framework. UC describes a real and an ideal world. Both are characterized by an execution of several interactive Turing machines (ITIs), depicted in Fig. 2. In the real world, this execution revolves around a (threshold signature) *protocol* Π-TSx (which we want to prove secure). We identify each party involved in the protocol by some party ID string $\mathcal{P}$ (called "pid" in [Can00]). For each party $\mathcal{P}$, we run an instance of Π-TSx, denoted Π-TSx$_{\mathcal{P}}$. These instances are stateful. They take direction from the *environment* $\mathcal{Z}$, which models all other protocols in the system, including ones that use Π-TSx as a subroutine.

Different instances Π-TSx$_{\mathcal{P}}$ do not interact directly. Instead, they share a communication channel functionality $\mathcal{F}$-Net (Functionality 1) and a key generation functionality $\mathcal{F}$-Gen as a shared (stateful) subroutine. This means that (network) messages from Π-TSx$_{\mathcal{P}_1}$ to Π-TSx$_{\mathcal{P}_2}$ are modeled by Π-TSx$_{\mathcal{P}_1}$ calling the Send procedure of $\mathcal{F}$-Net and Π-TSx$_{\mathcal{P}_2}$ later calling the Receive procedure of $\mathcal{F}$-Net (the exact working of the network functionality varies, see Functionalities 1 and 8). Different protocols Π-TSx may use different hybrid functionalities in place of or in addition to $\mathcal{F}$-Net, for example $\mathcal{F}$-Gen (Functionality 3) for trusted key generation.

Finally, adversarial behavior related to the protocol Π-TSx is modeled by an *adversary* $\mathcal{A}$. The adversary $\mathcal{A}$ also receives instructions from $\mathcal{Z}$ and gets to access $\mathcal{F}$-Net. Additionally, $\mathcal{F}$-Net allows $\mathcal{A}$ to delay, drop, and reorder messages.

In the ideal world, the protocol Π-TSx is replaced by an *ideal functionality* $\mathcal{F}$-TSx, which represents a secure-by-definition threshold signature functionality (in particular, when using $\mathcal{F}$-TSx, there exist no forgeries by definition). The adversary $\mathcal{A}$ is called $\mathcal{S}$ (for *simulator*) in the ideal world. The simulator's purpose is to translate any actions/attacks that $\mathcal{A}$ may execute in the real world into actions/attacks in the ideal world. The network functionality $\mathcal{F}$-Net vanishes in the ideal world. One may think of $\mathcal{F}$-Net as being absorbed into $\mathcal{S}$.

A protocol Π-TSx is said to be UC-secure if for every real world adversary $\mathcal{A}$, there exists an ideal world adversary (simulator) $\mathcal{S}$ such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with $\mathcal{A}$ and Π-TSx in the real world or with $\mathcal{S}$ and $\mathcal{F}$-TSx in the ideal world. Intuitively, this gives us security: any real-world attack by $\mathcal{A}$ can be translated into an ideal-world attack (where no meaningful attacks exist). Furthermore, it establishes composability: any protocol using the conveniently secure-by-definition functionality $\mathcal{F}$-TSx as a subroutine for its threshold signature needs can instead use any concrete secure protocol Π-TSx as a subroutine without any noticable difference.

Our notation for UC protocols and functionalities. Each box in Fig. 1 is (an instance of) an interactive Turing machine. These machines communicate via message passing (not to be confused with network messages). For example, the environment may send a message of the form (Sign, m) to the protocol Π-TSx$_{\mathcal{P}}$, which executes some computations, potentially a full protocol, and eventually then sends a message (Signature, m, σ) back to the environment.

In an effort to push UC protocol notation closer to game-based protocol notation [BDF+18], as well as overall improve notation, we describe our protocols using *procedure calls* rather than using message passing. For the example above, we would say that "the environment calls $\mathcal{P}$.Sign(m)", which "returns σ". Behind the scenes, these procedure calls are implemented using message passing.

We describe our UC pseudocode notation for an interactive Turing machine instance (ITI, e.g., Functionality 4) in the following. Each ITI has state, which is initialized in the special INIT procedure and accessible throughout all procedures of this ITI. The ITI declares several procedures in the following format: "$\mathcal{P}$.CommandName(*args*) [Code]", which translates to "when receiving message (CommandName, *cid*, *args*) with sender party $\mathcal{P}$, execute Code with input *args*"[6]. Here, *cid* is a unique identifier for this call, which is used to associate the response with the call. Code contains the usual instructions: reading and manipulating state, conditions, loops, etc. The instruction "**return** *result*" translates into "send message (✓CommandName, *cid*, *result*) to the caller of

[6] Additionally, if $\mathcal{P} \in \mathbf{H}$ is honest and the message comes from the UC adversary, it is ignored. If $\mathcal{P} \in \mathbf{C}$ is corrupt and the message is *not* from the UC adversary, it is ignored.

this procedure". Additionally, `Code` may itself call another procedure on another ITI, using an instruction of the form "$y \leftarrow$ `ITI`$::\mathcal{P}.\textsc{CommandName}'(args')$", which translates to "choose random unique identifier cid', send message $(\textsc{CommandName}', cid', args')$ to `ITI`, wait for a response message $(\checkmark\textsc{CommandName}', cid', result')$, assign $y := result'$, then continue executing `Code`".

The above describes (calls to) subroutines. A similar notation applies to procedures and calls involving the UC adversary $\mathcal{A}$ (called simulator $\mathcal{S}$ in the ideal world, this discussion applies to both). In particular, the instruction $y \leftarrow \clock\mathcal{A}::\textsc{CommandName}'(args')$ translates to "choose random unique identifier cid', send message $(\textsc{CommandName}', cid', args')$ to $\mathcal{A}$, and wait for a response message of the form $(\checkmark\textsc{CommandName}', cid', result')$, assign $y := result'$, then continue executing `Code`". Note the absence of a party identifier $\mathcal{P}$. Crucially, the adversary may choose not to send the response message immediately, but rather yield control to the environment, which may start new (concurrent) processes before returning. To emphasize this, we use the $\clock$ symbol for calls to the adversary. When writing UC pseudocode, this means that the calling ITI may have multiple instances of unfinished procedures waiting for a response. We use the $(\clock)$ symbol for procedures that call the adversary (directly or through one of their subroutines). Conversely, the absence of $(\clock)$ for a procedure indicates that it will return without it or any of its subroutines invoking the adversary.

In the real world, corruption for protocol ITI Π-$\mathtt{X}_{\mathcal{P}}$ is modeled by a procedure $\mathcal{A}.\textsc{Corrupt}()$, which can be called by the adversary. We explicitly model the behavior of this procedure in our protocols. We model static corruption by requiring that Π-$\mathtt{X}_{\mathcal{P}}::\mathcal{A}.\textsc{Corrupt}$ can only be called before Π-$\mathtt{X}_{\mathcal{P}}::\mathcal{P}.\textsc{KeyGen}$ is called (i.e., before any meaningful action has taken place for party $\mathcal{P}$). Generally, corrupting in a protocol does not reveal the subprotocols' states (however, the adversary can separately call Corrupt on the subprotocols to retrieve it). Corruption in the protocol does add the party $\mathcal{P}$ to the global $\mathbf{C}$ set, which means that subroutines will also consider $\mathcal{P}$ corrupted. In the ideal world, corruption of party $\mathcal{P}$ in the ideal functionality $\mathcal{F}$-`X` is modeled by a procedure $\mathcal{S}.\textsc{Corrupt}(\mathcal{P})$ (note that the UC adversary is called $\mathcal{S}$ in the ideal world). In both cases, upon corruption, party $\mathcal{P}$ is added to a globally readable set $\mathbf{C}$ of corrupted parties (which one may model using a global functionality $\mathcal{G}$-`Corruption`; we omit details). $\mathcal{P}$ is then considered corrupt for all protocols and functionalities, i.e., we consider party-wise corruption.

For initialization, if present, the special Init procedure runs once when the ITI is first created, i.e., when it receives its first message, before any other procedure's code is executed. Variables initialized in Init are accessible in all other procedures of the ITI. Other ITIs can optionally explicitly invoke Init. This returns immediately if Init has been called before (implicitly or explicitly), and otherwise this runs Init and returns (with not return value) once it finishes.

Finally, to enable multiple copies of a functionality or instance, each copy is associated with a session ID *sid*. We can imagine *sid* to be part of the subscript for instance Π-$\mathtt{X}_{sid,\mathcal{P}}$ and $\mathcal{F}$-$\mathtt{X}_{sid}$. However, we omit *sid* from notation for notational convenience. Only for the global clock functionality $\mathcal{G}$-`Clock` (Functionality 7), the *sid* of the caller does play a role, as $\mathcal{G}$-`Clock` maintains separate clocks for each session.

3 Game-based Definitions for Threshold Signatures

In this section, we present game-based definitions for threshold signature schemes (supporting various levels of preprocessing). We first introduce the syntax of such schemes, and then present definitions for correctness, robustness, and unforgeability.

3.1 Syntax of Threshold Signature Schemes

We define a *threshold signature scheme* Σ_{TS} as a tuple

$$(\mathbf{rnd}, \mathsf{Setup}, \mathsf{KeyGen}, \{\mathsf{Sign}_k\}_{k=1}^{\mathsf{rnd}_{\mathrm{sign}}}, \mathsf{Verify}, \mathsf{ssid})$$

Here, $\mathbf{rnd} = (\mathsf{rnd}_{\mathrm{T}}, \mathsf{rnd}_{\mathrm{msg}}, \mathsf{rnd}_{\mathrm{ctx}}, \mathsf{rnd}_{\mathrm{com}}, \mathsf{rnd}_{\mathrm{sign}})$ with $\mathsf{rnd}_{\mathrm{T}}, \mathsf{rnd}_{\mathrm{msg}}, \mathsf{rnd}_{\mathrm{ctx}} \leq \mathsf{rnd}_{\mathrm{com}} \leq \mathsf{rnd}_{\mathrm{sign}} \in \mathbb{N}$ and $\mathsf{rnd}_{\mathrm{T}} \leq 2$. $\mathsf{rnd}_{\mathrm{sign}}$ is the total number of rounds in the signing protocol; $\mathsf{rnd}_{\mathrm{T}}$ is the round in which the set of signers $\boldsymbol{T}$ is determined; In more detail, $\mathsf{rnd}_{\mathrm{msg}}$ is the round in which the message-to-sign m is provided; $\mathsf{rnd}_{\mathrm{ctx}}$ is the round in which the context string ctx is provided; and $\mathsf{rnd}_{\mathrm{com}} \geq \max(\mathsf{rnd}_{\mathrm{msg}}, \mathsf{rnd}_{\mathrm{T}}, \mathsf{rnd}_{\mathrm{ctx}})$ is the *committing round* in which parties commit to all the necessary inputs (i.e., $\boldsymbol{T}, m, ctx$) and consent to creating a signature. We introduce these parameters to

capture various preprocessing configurations in the literature. For instance, FROST [KG20, BCK+22] and EKT [EKT24, ZT25] have $\mathsf{rnd}_{\text{sign}} = 2$ and $\mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{com}} = 2$, meaning that the first round is entirely independent of the message and set of signers, while the second round takes both inputs. Schemes like Ringtail [BKL+25] ($\mathsf{rnd}_{\text{sign}} = 2$) and Snap [KRT24] ($\mathsf{rnd}_{\text{sign}} = 4$), whose signer set must be fixed at the beginning, have $\mathsf{rnd}_{\text{T}} = 1$ and $\mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{com}} = 2$, and Crackle [KRT24] ($\mathsf{rnd}_{\text{sign}} = 5$) has $\mathsf{rnd}_{\text{T}} = 2$, $\mathsf{rnd}_{\text{msg}} = 3$.

We now define the algorithms as follows:

$\underline{\mathsf{Setup}(1^\kappa) \to \mathsf{par}}$ takes as input a security parameter and outputs public parameters par.

$\underline{\mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t}) \to (\mathsf{pk}, (\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}})}$ takes as input public parameters par, a set $\mathbf{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}$ of party identifiers, and threshold parameters $\mathbf{t} = (t_{\text{correct}}, t_{\text{corrupt}}, t_{\text{UF}}, t_{\text{robust}})$, where t_{correct} is a *correctness threshold* (i.e., t_{correct} honest parties can generate a valid signature), t_{corrupt} is a *corruption threshold* (i.e., the minimum number of corrupt parties that voids the security, such as unforgeability and robustness), t_{UF} is an *unforgeability threshold* (i.e., t_{UF} honest and corrupt signers together may circumvent unforgeability), and t_{robust} is a *robustness threshold* (i.e., t_{robust} parties can generate a valid signature even in the presence of up to t_{corrupt} corrupt parties). For simplicity, one may think $t_{\text{correct}} = t_{\text{corrupt}} = t_{\text{UF}} = t_{\text{robust}}$, which is a typical case. We require $t_{\text{correct}}, t_{\text{robust}} \geq t_{\text{UF}} \geq 1$ and $t_{\text{robust}} \geq t_{\text{corrupt}}$. KeyGen outputs a public verification key pk and a list of secret signing keys $(\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}$. In the games introduced later, we derive initial party states $\mathsf{st}_\mathcal{P} = (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P})$ for each party $\mathcal{P} \in \mathbf{P}$.

$\underline{\mathsf{Sign}_k(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \mathbf{pm}_{k-1}) \to (\mathsf{st}'_\mathcal{P}, \mathsf{st}'_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}') \text{ for } k = 1, \ldots, \mathsf{rnd}_{\text{sign}}}$ takes as input a *global state* $\mathsf{st}_\mathcal{P}$, a *local state* $\mathsf{st}_{\mathcal{P},\mathsf{eid}}$, a set $\boldsymbol{T} \subseteq \mathbf{P}$ of signers, the message-to-sign m, a context string ctx, and, for $k > 1$, received protocol messages $\mathbf{pm}_{k-1} = (\mathsf{pm}_{\mathcal{Q},k-1})_{\mathcal{Q}\in\boldsymbol{T}}$. (We set $\mathbf{pm}_0 = ()$.) Here, $\mathsf{st}_\mathcal{P}$ is intended to capture long-term state that persists across multiple executions of the signing protocol, while $\mathsf{st}_{\mathcal{P},\mathsf{eid}}$ is intended to capture short-term state that is specific to a single execution, identified by *execution ID* eid. To retrieve party $\mathcal{P}$'s protocol message, we write $\mathbf{pm}_{k-1}[\mathcal{P}]$ and we assume it to return $\perp$ if the message from $\mathcal{P}$ is not present in $\mathbf{pm}_{k-1}$. It outputs a protocol message $\mathsf{pm}_{\mathcal{P},k}$ and updated states $\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}$. Note that this update does not imply secure erasure of state; these states typically store randomness and intermediate data generated during the protocol. To model a scheme with secure erasure, one can define $\mathsf{st}_{\mathcal{P},\mathsf{eid}}$ to be empty after a certain round. We note that $\boldsymbol{T}, m, ctx$ can be $\perp$ in early rounds, but are expected to be supplied starting in round $k = \mathsf{rnd}_{\text{T}}, \mathsf{rnd}_{\text{msg}}, \mathsf{rnd}_{\text{ctx}}$, respectively. (Early rounds having partial input models preprocessing.)

$\underline{\mathsf{Combine}(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]}) \to \sigma}$ takes a public key pk, a set of signing parties $\boldsymbol{T}$, a message m, a context string ctx, and protocol messages $(\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]}$ as input, and outputs a signature σ. We write trs as a shorthand for *protocol transcript* $(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]})$.

$\underline{\mathsf{Verify}(\mathsf{pk}, m, \sigma) \to b}$ is a deterministic algorithm that takes a verification key pk, a message m, and a purported signature σ as input, and outputs a bit b.

$\underline{\mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]}) \to ssid}$ is a deterministic algorithm that takes a set of signing parties $\boldsymbol{T}$, a message m, context string ctx, and protocol messages $(\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]}$ received in or before the committing round $\mathsf{rnd}_{\text{com}}$, and outputs a signing session ID $ssid$. We introduce this algorithm to capture granular hierarchies of unforgeability levels (see Section 3.4).

3.2 Correctness

We introduce a "UC-friendly" notion of correctness. While property-based notions like correctness and unforgeability are often individually defined in separate games exposing different interfaces to the adversary, we aim to define them in a way that is as close to the UC setting as possible, where those properties are defined as part of the *same* ideal functionality. As such, in our threshold correctness game (TCOR) presented in Game 1, we opt to grant the adversary access to all the oracles available in the robustness and unforgeability games defined later. Essentially, it allows an adversary to query signing oracles defined in Oracle 1 for multiple sessions, and then pick one of those sessions to win the game by ensuring that the combined protocol messages do not yield a valid signature. Depending on the corruption parameter specified as $\mathsf{cor} \in \{\text{static}, \text{adaptive}\}$, the adversary can also either select all corrupt parties before seeing the public key (static corruption) or adaptively corrupt parties at any time through $\mathcal{O}_{\text{COR}}$ (adaptive corruption). It is worth noting that the oracle $\mathcal{O}_{\text{SIGN},k}$ keeps track

Game 1: Correctness and Robustness Experiments

The code below references $\mathcal{O}_{\text{SIGN},k}, \mathcal{O}_{\text{COR}}$ from Oracle 1.

$\mathsf{Exp}^{\mathsf{cor},[\mathsf{TCOR}],k,[\mathsf{ROB}]}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t})$

1: $\text{SGN}[\cdot,\cdot] := \varnothing$
2: $\mathsf{par} \leftarrow \mathsf{Setup}(1^\kappa)$
3: $\mathbf{C} \leftarrow \mathcal{A}(\mathsf{par})$
4: **assert** $\mathbf{C} \subseteq \mathbf{P}$
5: $\mathbf{H} := \mathbf{P} \setminus \mathbf{C}$
6: $(\mathsf{pk}, (\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}) \leftarrow \mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})$
7: **for** $\mathcal{P} \in \mathbf{P}$ **do**
8: $\quad \mathsf{st}_\mathcal{P} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P})$
9: $\quad \text{ST}_\mathcal{P}[\mathsf{eid}] := (0, \bot, \ldots, \bot)$ for all eid
10: $\mathcal{O} := (\mathcal{O}_{\text{COR}}, (\mathcal{O}_{\text{SIGN},k})_{k\in[\mathsf{rnd}_{\text{sign}}]})$
11: $(\mathsf{pk}', \mathsf{trs}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{par}, \mathsf{pk}, \{\mathsf{sk}_\mathcal{P}\}_{\mathcal{P}\in\mathbf{C}})$
12: **assert** $\mathsf{ValidateC_t}(\mathsf{pk}', \mathsf{trs}) = 1$
13: **assert** $\mathsf{ValidateR_t}(\mathsf{pk}', \mathsf{trs}) = 1$
14: $\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}', \mathsf{trs})$
15: **return** $(\mathsf{Verify}(\mathsf{pk}', \mathsf{trs}.m, \sigma) = 0)$

$\mathsf{ValidateC_t}(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]})$

1: **assert** $\mathsf{pk}' = \mathsf{pk}$
2: **assert** $|\mathbf{C} \cap \boldsymbol{T}| = 0$
3: **assert** $|\boldsymbol{T}| \geq t_{\text{correct}}$
4: **assert** $\forall \mathcal{Q} \in \boldsymbol{T}\ \ \exists \mathsf{eid}_\mathcal{Q}$:
$\text{ST}_\mathcal{Q}[\mathsf{eid}_\mathcal{Q}] = (\mathsf{rnd}_{\text{sign}}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}-1]}, \mathbf{pm}_{\mathsf{rnd}_{\text{sign}}}[\mathcal{Q}], \cdot)$
// Input consistent with honest parties' views
5: **return** 1

$\mathsf{ValidateR_t}(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]})$

1: **assert** $\mathsf{pk}' = \mathsf{pk}$
2: **assert** $|\mathbf{C}| < t_{\text{corrupt}}$
3: **assert** $|\boldsymbol{T}| \geq t_{\text{robust}}$
4: $ssid := \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]})$
5: **assert** $\forall \mathcal{Q} \in \boldsymbol{T} \cap \mathbf{H}\ \ \exists \mathsf{eid}_\mathcal{Q}\ \ \exists (\mathbf{pm}'_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}-1]}$:
$\big(\text{ST}_\mathcal{Q}[\mathsf{eid}_\mathcal{Q}] = (\mathsf{rnd}_{\text{sign}}, \boldsymbol{T}, m, ctx, (\mathbf{pm}'_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}-1]}, \mathbf{pm}_{\mathsf{rnd}_{\text{sign}}}[\mathcal{Q}], \cdot)$
$\wedge \forall \ell \in [\mathsf{rnd}_{\text{sign}} - 1]\ \forall \mathcal{P} \in \boldsymbol{T} \cap \mathbf{H} : \mathbf{pm}'_\ell[\mathcal{P}] = \mathbf{pm}_\ell[\mathcal{P}]$
$\wedge \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}'_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]}) = ssid\big)$
// Protocol messages from honest parties and *ssid* are consistent (but parties may disagree on messages from corrupted parties)
6: **assert** $\exists \mathcal{Q} \in \boldsymbol{T} \cap \mathbf{H}\ \ \exists \mathsf{eid}_\mathcal{Q}$:
$\text{ST}_\mathcal{Q}[\mathsf{eid}_\mathcal{Q}] = (\mathsf{rnd}_{\text{sign}}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}-1]}, \mathbf{pm}_{\mathsf{rnd}_{\text{sign}}}[\mathcal{Q}], \cdot)$ // At least one honest party's view is consistent with the given input
7: **return** 1

of both the global state $\mathsf{st}_\mathcal{P}$ and session-specific states $\text{ST}_\mathcal{P}[\mathsf{eid}]$ for each party $\mathcal{P}$ and execution ID eid. This is to model the fact that parties may maintain long-term state across multiple executions, while also keeping short-lived state for each execution. The predicate $\mathsf{ValidateC_t}$ checking winning conditions makes sure that all parties in $\boldsymbol{T}$ are uncorrupted and have consistent, honestly generated views. While the oracle handles unforgeability bookkeeping in the committing round $\mathsf{rnd}_{\text{com}}$, the reader can safely ignore it as this will be only relevant for unforgeability.

Definition 1 (Correctness). *Let* $\mathsf{cor} \in \{\text{adaptive}, \text{static}\}$. *A threshold signature scheme* Σ_{TS} *is* $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-correct *if for any stateful ppt* $\mathcal{A}$, *the probability* $\Pr[\mathsf{Exp}^{\mathsf{cor},\mathsf{TCOR}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t}) = 1]$ *is negligible in* κ.

In the literature, however, correctness is often defined in a more restricted manner, where an adversary is only allowed to run a signing protocol once with fixed inputs $(\boldsymbol{T}, m, ctx)$ after seeing the public key. We call this *one-time correctness* (1TCOR):

Definition 2 (One-time Correctness). *A threshold signature scheme* Σ_{TS} *is* $(\mathbf{P}, \mathbf{t})$-one-time correct *if for any stateful ppt* $\mathcal{A}$, *the probability* $\Pr[\mathsf{Exp}^{\mathsf{1TCOR}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t}) = 1]$ *is negligible in* κ.

We show if a scheme fulfills 1TCOR perfectly and does not keep mutable state between different signing sessions (as is the case for many schemes in the literature), then it also fulfills TCOR.

Lemma 1. *Let* Σ_{TS} *be a scheme which does not update its global state* $\mathsf{st}_\mathcal{P}$. Σ_{TS} *is perfectly* $(\mathbf{P}, \mathbf{t})$-*one-time correct if and only if it is perfectly* $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-*correct for* $\mathsf{cor} \in \{\text{adaptive}, \text{static}\}$.

Proof. The "if" direction is trivial: given an adversary $\mathcal{A}$ against 1TCOR, one can obtain an adversary against TCOR that makes queries to the signing oracles with $(\boldsymbol{T}, m, ctx)$ output by $\mathcal{A}$. Clearly, the protocol messages derived from such queries lead to an invalid signature if they break 1TCOR.

We now show the "only if" direction. Since $\mathsf{rnd}_{\text{T}} \leq 2$, there is no way for an adversary to inject maliciously crafted protocol messages to the input of Sign_k as long as the conditions described in $\mathsf{ValidateC_t}$ hold. Moreover, since the global state $\mathsf{st}_\mathcal{P}$ remains unchanged,[7] per-execution state $\mathsf{st}_{\mathcal{P},\mathsf{eid}}$

[7] We note that this condition is crucial here. Consider a scheme that perfectly correctly signs the first time, then sets a flag in its global state, and declines to sign a second time. This scheme would be one-time correct, but clearly not correct in terms of Definition 1.

Game 2: One-time Correctness Experiment

$\mathsf{Exp}^{\mathsf{1TCOR}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t})$

$\mathsf{par} \leftarrow \mathsf{Setup}(1^\kappa)$
$(\mathsf{pk}, (\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}) \leftarrow \mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})$
for $\mathcal{P} \in \mathbf{P}$ **do**
  $\mathsf{st}_\mathcal{P} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P})$
  $\mathsf{st}_{\mathcal{P},\mathsf{eid}} := \bot$
$\mathbf{pm}_0 := (); \boldsymbol{T} := \bot; m := \bot; ctx := \bot$
$\mathsf{inp} \leftarrow \mathcal{A}(\mathsf{par}, \mathsf{pk})$
for $k = 1, \ldots, \mathsf{rnd}_{\mathrm{sign}}$ **do**
  if $k = \mathsf{rnd}_\mathrm{T}$ **then** $\boldsymbol{T} := \mathsf{inp}.\mathrm{T}$, **assert** $\boldsymbol{T} \subseteq \mathbf{P}$
  if $k = \mathsf{rnd}_{\mathrm{msg}}$ **then** $m := \mathsf{inp}.\mathrm{m}$
  if $k = \mathsf{rnd}_{\mathrm{ctx}}$ **then** $ctx := \mathsf{inp}.\mathrm{ctx}$
  for $\mathcal{P} \in \boldsymbol{T}$ **do**
    $(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}_{k,\mathcal{P}}) \leftarrow \mathsf{Sign}_k(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \mathbf{pm}_{k-1})$
  $\mathbf{pm}_k := (\mathsf{pm}_{k,\mathcal{P}})_{\mathcal{P}\in\boldsymbol{T}}$
$\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\mathrm{sign}}]})$
return $(\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 0 \wedge |\boldsymbol{T}| \geq t_{\mathrm{correct}})$

also remains unchanged regardless of other concurrent signing sessions. Therefore, assuming 1TCOR holds perfectly, each honest execution of the signing protocol always outputs a valid signature for any public key, key shares, $\boldsymbol{T}$, m, and ctx. The winning condition of Game 1 demands that the adversary select a single execution of the signing protocol with the same input ($\boldsymbol{T}, m, ctx$) and protocol messages. Thus, the combined protocol messages from such an execution should always output a valid signature. □

In the imperfect case, however, the two notions are not equivalent. This is mainly due to the fact that in TCOR an adversary can decide a message m after observing the output of Sign_1 if $\mathsf{rnd}_{\mathrm{msg}} > 1$.

Lemma 2. *There exists a threshold signature scheme that is statistically one-time correct, but not statistically correct.*

Proof. Consider a variant of FROST1 (Protocol 9) in which Sign_2 refuses to sign if it receives $m = R||S$ as a message to sign, where (R, S) has been previously output by Sign_1 in the same execution. Clearly, this scheme is statistically one-time correct since the probability that an adversary guesses such a message in advance is negligible. However, it is not statistically correct since an adversary can set $m = R||S$ after observing Sign_1's output, causing verification to fail with probability 1. □

3.3 Robustness

Following our interactive correctness definition, we define robustness in Game 1 similarly. The only difference with TCOR is the winning condition specified by $\mathsf{ValidateR}_\mathbf{t}$ as opposed to $\mathsf{ValidateC}_\mathbf{t}$. Essentially, the adversary wins if (1) the public key is valid, (2) the number of corruptions has not reached the corruption threshold t_{corrupt}, (3) the number of signers is at least the threshold t_{robust} (i.e., t_{robust} parties can generate a valid signature even in the presence of up to $t_{\mathrm{corrupt}} - 1$ corruptions), (4) all honest signers in the specified signing set $\boldsymbol{T}$ agree on *ssid* and have consistent views with the provided protocol messages from honest parties, and (5) the combined protocol messages do not yield a valid signature. To the best of our knowledge, [RRJ+22] and [ENP24] are the only works that define robustness in a game-based setting. Our definition is similar in spirit to [RRJ+22], where the adversary has control over the scheduling and delivery of messages and honest parties may not receive consistent messages from corrupt parties (and thus broadcast channels are not needed to ensure consistency). The main difference is that we do not assume an adversary to honestly relay messages between honest parties; rather, we impose this as a winning condition (4) in $\mathsf{ValidateR}_\mathbf{t}$. In contrast, the definition in [ENP24] assumes synchronous broadcast channels, since the oracle for signing protocols is defined in a way that all honest parties receive the same messages from corrupt parties and executed simultaneously.

Oracle 1: Oracles for Signing and Corruption

$\underline{\mathcal{O}_{\text{SIGN},k}(\mathcal{P}, \mathsf{eid}, \mathsf{inp}),\ k \in \{1, \ldots, \mathsf{rnd}_{\text{sign}}\}}$

// Retrieve state
$(\mathrm{rnd}_{\text{done}}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [k-2]}, \mathsf{pm}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}) := \mathrm{ST}_{\mathcal{P}}[\mathsf{eid}]$
assert $\mathrm{rnd}_{\text{done}} = k - 1$
// Take $\boldsymbol{T}, m, ctx$ from inp in prescribed rounds
if $k = \mathsf{rnd}_{\text{T}}$ **then** $\boldsymbol{T} := \mathsf{inp}.\text{T}$, **assert** $\{\mathcal{P}\} \subseteq \boldsymbol{T} \subseteq \mathbf{P}$
if $k = \mathsf{rnd}_{\text{msg}}$ **then** $m := \mathsf{inp}.\text{m}$
if $k = \mathsf{rnd}_{\text{ctx}}$ **then** $ctx := \mathsf{inp}.\text{ctx}$
if $k > 1$ **then** // Collect protocol messages in $\mathbf{pm}_{k-1}$
  $\mathbf{pm}_{k-1} := \mathsf{inp}.\mathbf{pm}$ // Other parties (from input)
  $\mathbf{pm}_{k-1}[\mathcal{P}] := \mathsf{pm}$ // $\mathcal{P}$'s sent message (from state)
// Unforgeability bookkeeping
if $k = \mathsf{rnd}_{\text{com}}$ **then**
  $ssid := \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [k-1]})$
  $\mathrm{SGN}[m, ssid] := \mathrm{SGN}[m, ssid] \cup \{\mathcal{P}\}$
// Run round function
$(\mathsf{st}'_{\mathcal{P}}, \mathsf{st}'_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}') \leftarrow \mathsf{Sign}_k(\mathsf{st}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \mathbf{pm}_{k-1})$
$\mathrm{rnd}_{\text{done}} := k$
// Store the updated state
$\mathrm{ST}_{\mathcal{P}}[\mathsf{eid}] := (\mathrm{rnd}_{\text{done}}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [k-1]}, \mathsf{pm}', \mathsf{st}'_{\mathcal{P},\mathsf{eid}})$
$\mathsf{st}_{\mathcal{P}} := \mathsf{st}'_{\mathcal{P}}$
return pm'

$\underline{\mathcal{O}_{\text{COR}}(\mathcal{P})}$

assert $\mathsf{cor} = \text{adaptive}$
if $(\mathcal{P} \notin \mathbf{H}) \vee (|\mathbf{C}| = t_{\text{corrupt}} - 1)$ **then**
  return $\perp$
$\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
return $(\mathsf{sk}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P}}, \mathrm{ST}_{\mathcal{P}})$

Definition 3 (Robustness). *Let* $\mathsf{cor} \in \{\text{adaptive}, \text{static}\}$. *A threshold signature scheme* Σ_{TS} *is* $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-robust *if for all stateful ppt* $\mathcal{A}$*, the probability* $\Pr[\mathsf{Exp}^{\mathsf{cor},\mathsf{ROB}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t}) = 1]$ *is negligible in* κ.

We note that, unlike correctness and unforgeability, robustness is an optional property that may not be required in all applications. To model schemes that do not provide robustness, we allow t_{robust} to be set to $|\mathbf{P}|+1$, in which case the adversary can never satisfy the winning condition (3) in $\mathsf{ValidateR}_{\mathbf{t}}$.

Given the limited literature on game-based definitions of robustness, we do not include robust schemes in Table 1. However, our definition gives a template for protocol designers to prove the robustness of schemes in the future.

Game 3: Unforgeability experiment for a threshold signature scheme

The code below references $\mathcal{O}_{\text{SIGN},k}, \mathcal{O}_{\text{COR}}$ from Oracle 1 and $\mathrm{Signed}_{\mathsf{lvl},t_{\text{UF}}}, \mathrm{NumSig}_{\mathsf{lvl},t_{\text{UF}}}$ from Table 3.

$\underline{\mathsf{Exp}^{\mathsf{cor},\mathsf{lvl},\mathsf{UF}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t})}$

$\mathrm{SGN}[\cdot,\cdot] := \varnothing$
$\mathsf{par} \leftarrow \mathsf{Setup}(1^\kappa)$
$\mathbf{C} \leftarrow \mathcal{A}(\mathsf{par})$
assert $\mathbf{C} \subseteq \mathbf{P}$
$\mathbf{H} := \mathbf{P} \setminus \mathbf{C}$
$(\mathsf{pk}, (\mathsf{sk}_{\mathcal{P}})_{\mathcal{P} \in \mathbf{P}}) \leftarrow \mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})$
for $\mathcal{P} \in \mathbf{P}$ **do**
  $\mathsf{st}_{\mathcal{P}} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_{\mathcal{P}})$
  $\mathrm{ST}_{\mathcal{P}}[\mathsf{eid}] := (0, \perp, \ldots, \perp)$ for all eid
$\mathcal{O} := (\mathcal{O}_{\text{COR}}, (\mathcal{O}_{\text{SIGN},k})_{k \in [\mathsf{rnd}_{\text{sign}}]})$
$(m^*, \sigma^*, (\sigma^*_i)_{i \in [\ell]}) \leftarrow \mathcal{A}^{\mathcal{O}}(\mathsf{par}, \mathsf{pk}, (\mathsf{sk}_{\mathcal{P}})_{\mathcal{P} \in \mathbf{C}})$
$\mathrm{VER}[\cdot,\cdot,\cdot] := \varnothing$
for $i \in [\ell]$ **do**
  $b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*_i)$
  $\mathrm{VER}[\mathsf{pk}, m^*, \sigma^*_i] := b$
  assert $\sigma^* \neq \sigma^*_i$
assert $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}, m^*) = 1$
return $(\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = 1)$

$\underline{\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}', m)}$

assert $\mathsf{pk}' = \mathsf{pk}$ // Verification key
assert $|\mathbf{C}| < t_{\text{corrupt}}$ // Corruption limit
if $\mathsf{lvl} = \text{UF-X}$ **then** // Unforgeability
  return $(\neg \mathrm{Signed}_{\mathsf{lvl},t_{\text{UF}}}(m))$
if $\mathsf{lvl} = \text{SUF-X}$ **then** // Strong UF
  return $(|\{\sigma' \mid \mathrm{VER}[\mathsf{pk}, m, \sigma'] = 1\}| \geq \mathrm{NumSig}_{\mathsf{lvl},t_{\text{UF}}}(m))$

3.4 Unforgeability

With the signing and corruption oracles defined in Oracle 1 at hand, we can now define various levels of unforgeability (see Game 3). The interfaces exposed to the adversary are identical to those in the correctness and robustness games. The difference lies in the winning condition. As the adversary can concurrently query the signing oracles for multiple sessions, one can consider different levels of security and their non-triviality checks, as observed in [BTZ22, CPZ24]. To capture the hierarchies of (strong) unforgeability levels, we introduce a parameter lvl, whose possible values and their interpretations are summarized in Table 3. At a high level, UF-X (unforgeability) considers any forgery on message m trivial if *a sufficiently large number of honest parties have previously signed* m, as specified by the predicate $\text{Signed}_{\text{UF-}X}(m)$. Depending on the level, the $\text{Signed}_{\text{UF-}X}(m)$ predicate checks different conditions on the set $\text{SGN}[m, ssid]$ of honest parties that have executed the committing round $\mathsf{rnd}_{\text{com}}$ for m in a signing session identified by *ssid* (defined in the next paragraph). The adversary then wins if a signature σ^* is valid with respect to m^* and the predicate $\text{Signed}_{\text{UF-}X}(m^*)$ does *not* hold. For strong unforgeability SUF-X, we follow a *one-more* unforgeability style of definition, similar to [NT24, LNÖ25] (though significantly stronger than [NT24]). Table 3 defines $\text{NumSig}_{\text{SUF-}X,t_{\text{UF}}}(m)$, which tells us how many signatures are trivial to produce given the signing queries made. The adversary wins if it produces more than $\text{NumSig}_{\text{SUF-}X,t_{\text{UF}}}(m^*)$ different valid signatures on m^*. This generalizes the approach of [BTZ22, CPZ24], where for strong unforgeability, the rather strong restriction is for a scheme to require existentially *unique* signatures for every $m, ssid$. Because this is an information-theoretic existence property, it cannot be modeled as a check in a UC functionality, hence the need to switch to a one-more style definition. An additional benefit of this decision is that this definition is more relaxed and may encompass additional schemes not covered by the strong uniqueness requirement of [BTZ22, CPZ24].

Definition 4 (Unforgeability). *Let* $\mathsf{cor} \in \{\text{adaptive}, \text{static}\}$ *and let* lvl *be one of the parameters listed in Table 3. A threshold signature scheme* Σ_{TS} *is* $(\mathsf{cor}, \mathsf{lvl}, \mathbf{P}, \mathbf{t})$-unforgeable *if for all stateful ppt* $\mathcal{A}$, $\Pr[\mathsf{Exp}^{\mathsf{cor},\mathsf{lvl},\mathsf{UF}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t}) = 1]$ *is negligible in* κ.

The role of the signing session ID (*ssid*). Any signing process involves some metadata (information about the process apart from the message m to be signed). We call this extra metadata *ssid* (signing session ID). The general expectation is that signers need to agree on m and on *ssid* in order to produce a signature. Higher levels of unforgeability (Table 3) enforce this agreement: In UF-2 and above, asking t_{UF} parties to sign m in *different* sessions will *not* count as having requested a signature (w.r.t. the forgery non-triviality check). In practice, this means that signers would compute *ssid* before running the $\mathsf{rnd}_{\text{com}}$-th round of the signing protocol, inspect the information included in *ssid*, and abort if they are uncomfortable with creating a signature for that session. For example, *ssid* may contain some evidence *ctx* that signing is okay. For UF-2 and above, only if all signers received the same evidence and all signers agree with this evidence, a signature will actually materialize.

Additionally, session IDs play a crucial role in strong unforgeability SUF-X (Table 3). The general expectation for strong unforgeability is that for every $m, ssid$, at most one signature is going to be produced. Again, signers can inspect *ssid* before executing the committing round $\mathsf{rnd}_{\text{com}}$. If they do not want to produce, say, a second signature on m, they would decline additional signing processes with an *ssid* that is different from the *ssid* of the first signature.

We allow to parameterize the precise makeup of *ssid* via a function ssid that takes as input the signer set $\boldsymbol{T}$, the message m, additional (user-defined) context *ctx*, and the protocol messages $(\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]}$ from the rounds before the committing round $\mathsf{rnd}_{\text{com}}$ (i.e., the round in which parties commit to producing a signature). Note that $\mathsf{ssid}()$ is security-relevant: Generally, the more information ssid includes in *ssid*, the more information needs to be agreed upon by honest signers in order to produce a signature, the more different types of queries we allow the adversary without triggering a trivial forgery, the weaker the unforgeability guarantee.
We now list some example choices for $\mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]})$.

- $\mathsf{ssid}(\cdot) = 0$ (constant). This is the strongest notion. It means that *ssid* is essentially ignored. For strong unforgeability, this means that for any message m, there will never be more than one signature. BLS signatures [BLS04] are an example.
- $\mathsf{ssid}(\cdot) = (\boldsymbol{T}, ctx)$ (i.e. *ssid* is comprised of the signer set $\boldsymbol{T}$ and user-supplied context *ctx*). This is a flexible notion where the user defines the session. All signers need to agree with $\boldsymbol{T}$ and *ctx* in order to produce a signature.

– $\mathsf{ssid}(\cdot) = (\boldsymbol{T}, ctx, \mathbf{pm}_1)$ (i.e. *ssid* is comprised of $\boldsymbol{T}$, *ctx*, and the first round protocol messages $\mathbf{pm}_1$ (usually preprocessing)). This recovers the definitions of [BTZ22, BCK+22, CPZ24], where their *leader request* corresponds to our *ssid*. FROST [KG20, BCK+22] is an example. Note that our definition is flexible enough that it also captures strongly unforgeable schemes without preprocessing (e.g., $\mathsf{rnd}_{\text{com}} = 2$ while $\mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{ctx}} = 1$).

The levels of unforgeability. Following [BTZ22, BCK+22, CPZ24], we define different levels lvl of unforgeability. For this, Table 3 defines the predicate $\text{Signed}_{\mathsf{lvl},t_{\text{UF}}}(m)$, which tells us *whether*

Table 3: Different levels of security and their nontriviality checks, generalized from [BTZ22, CPZ24]. Here, "$\mathsf{ssid}(\cdot)$ includes $\boldsymbol{T}$ and $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$" means that $\mathsf{rnd}_{\text{com}} \geq 2$ and $\forall \boldsymbol{T}, (\mathbf{pm}_k)_k : \ \mathsf{ssid}(\boldsymbol{T}, \cdot, (\mathbf{pm}_k)_{k=1}^{\mathsf{rnd}_{\text{com}}-1}) = (\boldsymbol{T}, \mathbf{pm}_1, \cdot)$, i.e., ssid contains the signer set $\boldsymbol{T}$ and the first round messages $\mathbf{pm}_1$, which we call "preprocessing $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$" for easier comparison with [BTZ22, CPZ24]. "$\mathcal{Q}$ output $\mathsf{pp}_{\mathcal{Q}}$" means that $\mathsf{pp}_{\mathcal{Q}}$ was at some point output by a $\mathsf{Sign}_1(\mathcal{Q}, \dots)$ call (in Game 3 and Functionality 4) or output by a $\mathcal{Q}.\mathsf{Preprocess}(\dots)$ call (in Functionalities 5 and 9), i.e., the preprocessing specified by the adversary for $\mathcal{Q}$ was actually generated by $\mathcal{Q}$.

lvl	**Philosophy** What does it take to (potentially) produce a signature?	m **already queried?** $\text{Signed}_{\mathsf{lvl},t_{\text{UF}}}(m)$ ($\mathcal{A}$ wins if it produces a signature on m s.t. $\neg\text{Signed}_{\mathsf{lvl},t_{\text{UF}}}(m)$)
UF-0	At least one honest party must sign m	$\exists ssid : \ \text{SGN}[m, ssid] \cap \mathbf{H} \neq \varnothing$
UF-1	At least $t_{\text{UF}} - \lvert\mathbf{C}\rvert$ honest parties must sign m	$\left\lvert \bigcup_{ssid} \text{SGN}[m, ssid] \cup \mathbf{C} \right\rvert \geq t_{\text{UF}}$
UF-2	At least $t_{\text{UF}} - \lvert\mathbf{C}\rvert$ honest parties agree on $(m, ssid)$	$\exists ssid : \ \lvert\text{SGN}[m, ssid] \cup \mathbf{C}\rvert \geq t_{\text{UF}}$
UF-3	ssid includes signer set $\boldsymbol{T}$ and preprocessing $\mathsf{pp}_{\mathcal{Q}}$, and at least $t_{\text{UF}} - \lvert\mathbf{C}\rvert$ honest parties agree on $(ssid, m)$, and if $\mathsf{pp}_{\mathcal{Q}}$ is valid, then $\mathcal{Q}$ must agree.	$\mathsf{ssid}(\cdot)$ includes $\boldsymbol{T}$ and $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$. Furthermore, $\exists ssid = (\boldsymbol{T}, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}, \cdot) :$ $\lvert\text{SGN}[m, ssid] \cup \mathbf{C}\rvert \geq t_{\text{UF}}$ $\wedge \text{SGN}[m, ssid] \supseteq \{\mathcal{Q} \in \boldsymbol{T} \cap \mathbf{H} \mid \mathcal{Q} \text{ output } \mathsf{pp}_{\mathcal{Q}}\}$
UF-4	ssid includes signer set $\boldsymbol{T}$ and preprocessing, and all honest $\mathcal{P} \in \boldsymbol{T}$ must agree on $(ssid, m)$.	$\mathsf{ssid}(\cdot)$ includes $\boldsymbol{T}$ and $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$. Furthermore, $\exists ssid = (\boldsymbol{T}, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}, \cdot) :$ $\lvert\text{SGN}[m, ssid] \cup \mathbf{C}\rvert \geq t_{\text{UF}}$ $\wedge \text{SGN}[m, ssid] \cup \mathbf{C} \supseteq \boldsymbol{T}$

lvl	**Philosophy** We count the number of $(m, ssid)$ where ...	***How many* signatures on m queried?** $\text{NumSig}_{\mathsf{lvl},t_{\text{UF}}}(m)$ ($\mathcal{A}$ wins if it produces $> \text{NumSig}_{\mathsf{lvl},t_{\text{UF}}}(m)$ signatures on m)
SUF-2	At least $t_{\text{UF}} - \lvert\mathbf{C}\rvert$ honest parties agree on $(m, ssid)$	$\lvert\{ssid : \ \lvert\text{SGN}[m, ssid] \cup \mathbf{C}\rvert \geq t_{\text{UF}}\}\rvert$
SUF-3	ssid includes signer set $\boldsymbol{T}$ and preprocessing $\mathsf{pp}_{\mathcal{Q}}$, and at least $t_{\text{UF}} - \lvert\mathbf{C}\rvert$ honest parties agree on $(ssid, m)$, and if $\mathsf{pp}_{\mathcal{Q}}$ is valid, then $\mathcal{Q}$ must agree.	0 unless $\mathsf{ssid}(\cdot)$ includes $\boldsymbol{T}$ and $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$. If so: $\left\lvert \left\{ ssid : \begin{array}{c} ssid = (\boldsymbol{T}, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}, \cdot) \\ \wedge \lvert\text{SGN}[m, ssid] \cup \mathbf{C}\rvert \geq t_{\text{UF}} \\ \wedge \text{SGN}[m, ssid] \supseteq \\ \{\mathcal{Q} \in \boldsymbol{T} \cap \mathbf{H} \mid \mathcal{Q} \text{ output } \mathsf{pp}_{\mathcal{Q}}\} \end{array} \right\} \right\rvert$
SUF-4	ssid includes signer set $\boldsymbol{T}$ and preprocessing, and all honest $\mathcal{P} \in \boldsymbol{T}$ must agree on $(ssid, m)$.	0 unless $\mathsf{ssid}(\cdot)$ includes $\boldsymbol{T}$ and $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$. If so: $\left\lvert \left\{ ssid : \begin{array}{c} ssid = (\boldsymbol{T}, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}, \cdot) \\ \wedge \lvert\text{SGN}[m, ssid] \cup \mathbf{C}\rvert \geq t_{\text{UF}} \\ \wedge \text{SGN}[m, ssid] \cup \mathbf{C} \supseteq \boldsymbol{T} \end{array} \right\} \right\rvert$

we should expect a valid signature on m to exist (in a way we would not consider a forgery), and the function $\text{NumSig}_{\mathsf{lvl},t_{\text{UF}}}(m)$, which tells us *how many* signatures on m we should expect to exist. These are used in the triviality checks in Game 3. $\text{Signed}_{\mathsf{lvl},t_{\text{UF}}}$ and $\text{NumSig}_{\mathsf{lvl},t_{\text{UF}}}$ depend on the bookkeeping $\text{SGN}[m, ssid]$, which is the set of signers $\mathcal{P}$ that agreed to execute the committing round $\mathsf{rnd}_{\text{com}}$ for message m and session *ssid*. Note that for $i \in \{2, 3, 4\}$, we have $\text{Signed}_{\text{UF-}i}(m) \Leftrightarrow$

$\mathrm{NumSig}_{\text{SUF-}i}(m) > 0$, so unforgeability considers a signature on message m a forgery if and only if strong unforgeability would consider a single signature on m to be a forgery. Further note that there is no SUF-0 or SUF-1 level (consistent with [BTZ22,BCK+22]): The levels UF-0, UF-1 are characterized by effectively ignoring *ssid*. However, for a strong unforgeability notion, we need *ssid* to meaningfully group signing processes together and limit the number of expected signatures $\mathrm{NumSig}_{\mathsf{lvl},t_{\mathrm{UF}}}$ to the number of different *ssid*. See Appendix C for more discussion.

If we choose $\mathsf{ssid}(\cdot) = (\boldsymbol{T}, ctx, \mathbf{pm}_1)$, then our notions correspond to the ones in [BTZ22,BCK+22, CPZ24], with minor differences: Our security game (Game 3) rejects queries for honest parties $\mathcal{P}$ that are not part of the signer set $\boldsymbol{T}$, and does not allow queries where a party $\mathcal{P}$ is presented with a protocol message (e.g., preprocessing) $\mathsf{pm}_{\mathcal{P}}$ that it has not sent. In [BTZ22, BCK+22, CPZ24], it is the responsibility of the *construction* (rather than the game) to ensure such queries are rejected. Strictly speaking, this makes their notion stronger than ours, but the difference consists only of trivial checks. Our choice to have the game handle these trivial checks allows us to simplify the conditions in Table 3.

Prior work only considered $\mathsf{ssid}(\cdot) = (\boldsymbol{T}, ctx, \mathbf{pm}_1)$, or $\mathsf{ssid}(\cdot) = (\boldsymbol{T}, ctx)$ for schemes without preprocessing. In particular, this puts threshold BLS [Bol03] on hierarchy level UF-1 (under the t_{correct}-VCDH assumption [BTZ22, BCK+22], originally proven UF-0 under the GDH assumption in [Bol03]). This is mostly because threshold BLS does not meaningfully enforce any agreement on $ssid = (\boldsymbol{T}, ctx)$, which is needed for UF-2 and SUF-2. However, arguably, threshold BLS should be considered strongly unforgeable, as it even has unique signatures per message. We consider it a weakness of prior definitions that they cannot identify threshold BLS as strongly unforgeable. In our framework, setting $\mathsf{ssid}(\cdot) = 0$ for threshold BLS (i.e., threshold BLS does not promise to inspect any metadata before signing) allows us to put it at SUF-2, which means that we meaningfully capture its strong unforgeability property.

Theorem 1 (Threshold BLS is SUF-2 with $\mathsf{ssid}(\cdot) = 0$). *Consider Σ_{TS} with $\mathsf{ssid}(\cdot) = 0$ and unique signatures (i.e., for every pk, m, there exists at most one σ such that $\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 1$). If Σ_{TS} is $(\mathsf{cor}, \text{UF-1}, \mathbf{P}, \mathbf{t})$-unforgeable, then it is also $(\mathsf{cor}, \text{SUF-2}, \mathbf{P}, \mathbf{t})$-unforgeable.*

Note that the precondition applies to threshold BLS assuming the t_{correct}-vector CDH assumption, as it is known to be UF-1 [Gro21, BTZ22, BCK+22] in the static corruption setting with respect to the UF-1 definition of [BTZ22, BCK+22], which is equivalent to ours (with arbitrary ssid, as UF-1 is agnostic to ssid).

Proof. For $\mathsf{ssid}(\cdot) = 0$, if Σ_{TS} is $(\mathsf{cor}, \text{UF-1}, \mathbf{P}, \mathbf{t})$-unforgeable, then it is also $(\mathsf{cor}, \text{UF-2}, \mathbf{P}, \mathbf{t})$-unforgeable (cf. trivial forgery conditions in Table 3). If Σ_{TS} has unique signatures and is $(\mathsf{cor}, \text{UF-2}, \mathbf{P}, \mathbf{t})$-unforgeable, then it is also $(\mathsf{cor}, \text{SUF-2}, \mathbf{P}, \mathbf{t})$-unforgeable.

3.5 The interplay between correctness, robustness, and unforgeability

There is an inherent tension between correctness/robustness and unforgeability. We need to avoid the contradictory situation where correctness or robustness demand that honest users output a valid signature while unforgeability demands that no valid signature exists. This will be particularly relevant for Section 4, where an ideal functionality needs to enforce correctness, robustness, and unforgeability at the same time.

Consider the situation where $\mathsf{ValidateC}_{\mathbf{t}}(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]})$ outputs 1, i.e., the signature output by $\mathsf{Combine}$ must be valid due to correctness. In this case, we have that all honest signers $\mathcal{P} \in \boldsymbol{T}$ finished the signing process for $\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]}$. In particular, all parties agree on $ssid := \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]})$. Hence each $\mathcal{P} \in \boldsymbol{T}$ was added to the same $\mathrm{SGN}[m, ssid]$ during $\mathsf{rnd}_{\text{com}}$ of signing. As a consequence, note that $|\mathrm{SGN}[m, ssid] \cup \mathbf{C}| = |\mathrm{SGN}[m, ssid]| = |\boldsymbol{T}| \geq t_{\text{correct}} \geq t_{\mathrm{UF}}$ as required for most levels of unforgeability. By inspection of Table 3, this implies that $\mathrm{Signed}_{\mathsf{lvl},t_{\mathrm{UF}}}(m) = 1$ and $\mathrm{NumSig}_{\mathsf{lvl},t_{\mathrm{UF}}}(m) > 0$ for all lvl and $t_{\mathrm{UF}} \leq t_{\text{correct}}$. Hence the existence of one valid signature on m is not considered a violation of unforgeability.

Similarly, consider the situation where $\mathsf{ValidateR}_{\mathbf{t}}(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{sign}}]})$ outputs 1, i.e., the signature output by $\mathsf{Combine}$ must be valid due to robustness. This means that all honest parties $\mathcal{P} \in \boldsymbol{T} \cap \mathbf{H}$ finished the signing process for the same $m, ssid$ (honest parties agreeing on *ssid* is an explicit requirement of $\mathsf{ValidateR}_{\mathbf{t}}$). Hence, each $\mathcal{P} \in \boldsymbol{T} \cap \mathbf{H}$ was added to the same set $\mathrm{SGN}[m, ssid]$

Functionality 4: Fine-grained $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$

The code below references $\mathsf{ValidateC_t}$, $\mathsf{ValidateR_t}$ from Game 1, $\mathcal{O}_{\text{SIGN},k}$ from Oracle 1, and $\mathsf{ValidateUF_{lvl,t}}$ from Game 3.

(⏲)INIT() // Invoked once upon creation
1: $\text{VER}[\cdot,\cdot,\cdot] := \bot$
2: **for** $\mathcal{P} \in \mathbf{P}$ **do**
3: $\quad \text{ST}_{\mathcal{P}}[\mathsf{eid}] := (0, \bot, \ldots, \bot)$ for all eid
4: $\quad \text{Ready}[\mathcal{P}] := 0$
5: $\text{SGN}[\cdot,\cdot] := \varnothing$
6: $\mathsf{Sign}_k := (\cdot \mapsto (1,1,1))$ for $k \in [\mathsf{rnd}_{\text{sign}}]$ // Default
7: $\mathsf{Combine} := (\cdot \mapsto 1)$ // Default
8: $\mathsf{Verify} := (\cdot \mapsto 0)$ // Default
9: $\mathsf{pk} := \bot$
10: $\mathsf{st}_{\mathcal{P}} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_{\mathcal{P}} := 1)$ for all $\mathcal{P} \in \mathbf{P}$
11: $(\mathsf{Verify}, (\mathsf{Sign}_k)_{k=1}^{\mathsf{rnd}_{\text{sign}}}, \mathsf{Combine}, \mathsf{pk}', (\mathsf{st}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}})_{\mathcal{P}\in\mathbf{P}}) \leftarrow$ ⏲$\mathcal{S}$::GETALG() // Interpret $\mathsf{pk}' \neq \bot$
12: **if** $\mathsf{pk} = \bot$ **then**
13: $\quad \mathsf{pk} := \mathsf{pk}'$

$\mathcal{P}$.KEYGEN() for $\mathcal{P} \in \mathbf{P}$
14: **if** $\mathsf{pk} = \bot$ **then** $\mathsf{pk} := 1$ // Default if $\mathcal{S}$ has not set pk.
15: $\text{Ready}[\mathcal{P}] := 1$
16: **return** pk

$\mathcal{S}$.CORRUPT($\mathcal{P}$)
17: **assert** $\mathcal{P} \in \mathbf{P}$
18: **assert** $\mathsf{cor} = \text{adaptive} \vee$ [$\mathcal{P}$.KEYGEN has not been called yet]
19: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
20: **return** $(\mathsf{sk}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P}}, \text{ST}_{\mathcal{P}})$

$\mathcal{P}$.SIGN$_k$(eid, inp) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}, k \in [\mathsf{rnd}_{\text{sign}}]$
21: **assert** $\text{Ready}[\mathcal{P}] = 1$
22: **return** $\mathcal{O}_{\text{SIGN},k}(\mathcal{P}, \mathsf{eid}, \mathsf{inp})$ // See Oracle 1.

$\mathcal{V}$.VERIFY(pk', m, σ) for any honest party $\mathcal{V} \notin \mathbf{C}$
23: $b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
24: **if** $\mathsf{ValidateUF_{lvl,t}}(\mathsf{pk}', m)$ **then**
25: $\quad b := 0$ // Unforgeability
26: **if** $\text{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
27: $\quad \text{VER}[\mathsf{pk}', m, \sigma] := b$ // Consistent VERIFY output
28: **return** $\text{VER}[\mathsf{pk}', m, \sigma]$

$\mathcal{P}$.COMBINE($\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]}$) for honest $\mathcal{P} \notin \mathbf{C}$
29: $\mathsf{trs} := (\boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$
30: $\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}', \mathsf{trs})$
31: **if** $\mathsf{ValidateC_t}(\mathsf{pk}', \mathsf{trs})$ or $\mathsf{ValidateR_t}(\mathsf{pk}', \mathsf{trs})$ **then**
32: $\quad \sigma \leftarrow \mathsf{ForceValidity}(m, \sigma)$ // Correctness/Robustness
33: **return** σ

ForceValidity(m, σ) (internal)
34: **if** $\text{VERIFY}(\mathsf{pk}, m, \sigma) = 0$ **then**
35: $\quad$ // Declare some σ' valid if no valid signature exists.
36: $\quad$ **if** $\forall \sigma' :\ \text{VER}[\mathsf{pk}, m, \sigma'] \neq 1$ **then**
37: $\quad\quad \sigma \leftarrow_{\text{first}} \{\sigma' \in \{0,1\}^* \mid \text{VER}[\mathsf{pk}, m, \sigma'] = \bot\}$
38: $\quad\quad \text{VER}[\mathsf{pk}, m, \sigma] := 1$
39: $\quad$ // Replace σ with some valid signature.
40: $\quad \sigma \leftarrow_{\text{first}} \{\sigma' \in \{0,1\}^* \mid \text{VER}[\mathsf{pk}, m, \sigma'] = 1\}$
41: **return** σ

during $\mathsf{rnd}_{\text{com}}$ of signing, i.e., $\text{SGN}[m, ssid] \supseteq \boldsymbol{T} \cap \mathbf{H}$. As a consequence, note that $|\text{SGN}[m, ssid] \cup \mathbf{C}| \geq |(\boldsymbol{T} \cap \mathbf{H}) \cup \mathbf{C}| \geq |\boldsymbol{T}| \geq t_{\text{robust}} \geq t_{\text{UF}}$ as required for most levels of unforgeability. By inspection of Table 3, this implies that $\text{Signed}_{\mathsf{lvl},t_{\text{UF}}}(m) = 1$ and $\text{NumSig}_{\mathsf{lvl},t_{\text{UF}}}(m) > 0$ for all lvl and $t_{\text{UF}} \leq t_{\text{robust}}$. Hence the existence of one valid signature on m is not considered a violation of unforgeability.

4 $\mathcal{F}$-TS3: Fine-grained Threshold Signature Functionality

With the definitions of correctness, robustness and unforgeability in place, we can now define the fine-grained threshold signature functionality $\mathcal{F}$-TS3 (Functionality 4) that naturally captures these properties simultaneously. The spirit of $\mathcal{F}$-TS3 is to provide the environment $\mathcal{Z}$ with the same interface as the adversary $\mathcal{A}$ in Game 1 and 3. The functionality is equipped with interfaces that map to the algorithms and oracles of a threshold signature scheme Σ_{TS}. This includes a KEYGEN interface that returns the public key, a signing interface SIGN$_k$ for each round k that returns the protocol messages to be sent, a COMBINE interface that takes the transcript and outputs a final signature, and a signature verification interface VERIFY. Note that COMBINE and VERIFY can be called by any party, not just by the signing parties $\mathcal{P} \in \mathbf{P}$. The crucial difference is that $\mathcal{F}$-TS3 enforces correctness, robustness, and unforgeability by design, while in the games these properties are only guaranteed if the scheme Σ_{TS} satisfies the respective definitions. To model unforgeability, the VERIFY interface of $\mathcal{F}$-TS3 checks the predicate $\mathsf{ValidateUF_{lvl,t}}$ before returning the result of the verification algorithm Verify, and forces the decision bit to 0 if the predicate holds. Similarly, the COMBINE interface checks the predicates $\mathsf{ValidateC_t}$ and $\mathsf{ValidateR_t}$ before returning the output of the combining algorithm Combine, and forces the output to be a valid signature if either predicate holds.

Design philosophy and goals. When designing any ideal functionality $\mathcal{F}$, we need to be aware of the two ways it will be used: (1) For proving some protocol Π UC-realizes $\mathcal{F}$. To enable this use case, $\mathcal{F}$ must afford the ideal adversary $\mathcal{S}$ enough power to make real and ideal world indistinguishable. In this setting, we can informally think of $\mathcal{S}$ (which we construct) as being well-behaved. (2) As a subroutine of another protocol Π'. To enable this use case, $\mathcal{F}$ must provide a reasonable interface and

give strong guarantees to Π' in the presence of an *arbitrary* (not well-behaved) adversary $\mathcal{S}$. While both use cases are important, most of our discussion in the following revolves around the second use case, i.e., we imagine $\mathcal{F}$ to be used as a subroutine with an arbitrary adversary $\mathcal{S}$. We briefly touch on the first use case later when we discuss realizing the functionality.

When designing $\mathcal{F}$-TS3 specifically, we pursue three goals: (G1) Present a straightforward adaptation of the game-based definition into UC. (G2) Cover all properties (unforgeability, correctness, robustness) covered by the game-based definition. (G3) The procedures of $\mathcal{F}$-TS3 shall not yield to the adversary.

A straightforward adaptation of the game-based definition. For Goal G1, note that the SIGN_k procedure of $\mathcal{F}$-TS3 directly uses the signing oracle $\mathcal{O}_{\text{SIGN},k}$ and they share the same bookkeeping via $\text{ST}_{\mathcal{P}}$, VER, and SGN. The winning condition $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ from the unforgeability game is directly used in Line 24 to declare potential forgeries to be invalid signatures according to VERIFY. The winning conditions $\mathsf{ValidateC}_{\mathbf{t}}$ and $\mathsf{ValidateR}_{\mathbf{t}}$ from the correctness and robustness games are directly used in Line 31 to ensure that if the output of COMBINE *should* be a valid signature, ForceValidity ensures that a valid signature will be output. This makes $\mathcal{F}$-TS3 a straightforward adaptation of the game-based definitions into UC: The functionality uses the same bookkeeping and honest user behavior but effectively ensures that the winning conditions can never be fulfilled by rewriting the results of VERIFY and COMBINE in those cases where some winning condition would otherwise be fulfilled.

Care must be taken that enforcing correctness and robustness does not interfere with enforcing unforgeability. Inherently, there is a possible tension between these: Correctness and robustness require some signature be declared valid, while (strong) unforgeability imposes limits on whether (or how many) signatures can be considered valid. When correctness or robustness must be enforced, it happens via ForceValidity as described above. If the given σ is invalid (according to VERIFY), but it should be valid (according to $\mathsf{ValidateC}_{\mathbf{t}}$ or $\mathsf{ValidateR}_{\mathbf{t}}$), then $\mathsf{ForceValidity}(m, \sigma)$ is designed to output some *existing* valid signature σ' with $\text{VER}[\mathsf{pk}, m, \sigma'] = 1$ instead. Only if no such valid signature exists, ForceValidity declares the first bit string—that has not been rejected by VERIFY yet—to be a valid signature. This design ensures that ForceValidity never increases the number of valid signatures on a message m (which is important for strong unforgeability), except possibly from 0 to 1. As discussed in Section 3.5, if $\mathsf{ValidateC}_{\mathbf{t}}$ or $\mathsf{ValidateR}_{\mathbf{t}}$ demand that there must exist some valid signature on m, then $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ will tolerate existence of (at least one) valid signature on m. Hence going from 0 to 1 valid signatures through ForceValidity only happens in cases where this is compatible with our unforgeability expectations.

Explicit protocol messages pm**.** Goal G2 informs our choice not to hide the protocol messages in $\mathcal{F}$-TS3. The way $\mathcal{F}$-TS3 works is that the environment (/caller protocol) calls $\mathcal{P}.\text{SIGN}_k$ each round, which outputs the protocol message $\mathsf{pm}'_{\mathcal{P}}$ that $\mathcal{P}$ wants to send next. It is the environment's responsibility to deliver protocol messages to the other parties $\mathcal{P}'$ and pass them as part of inp to $\mathcal{P}'.\text{SIGN}_{k+1}$. This is unusual: for most ideal functionalities that model interactive protocols, protocol messages only exist in real protocols and are completely absent from the ideal functionality. Instead, such functionalities would just tell the adversary/simulator $\mathcal{S}$ about the signing request and wait for $\mathcal{S}$ to decide when to output the resulting final signature. Indeed, this is our approach for $\mathcal{F}$-TS2 and $\mathcal{F}$-TS1 (Sections 5 and 6). However, in order for $\mathcal{F}$-TS3 to model correctness and robustness, which essentially state some form of "if protocol messages are delivered correctly, then the resulting signature must be valid" requirement, $\mathcal{F}$-TS3 needs to keep track of protocol messages (in other words, $\mathsf{ValidateC}_{\mathbf{t}}, \mathsf{ValidateR}_{\mathbf{t}}$ require access to the protocol messages in ST).

No yielding to the adversary. Finally, Goal G3 is motivated by issues identified in [CDL+25] with functionalities that yield to the adversary during their execution: The calling protocol is effectively interrupted and gets no guarantees about when, if ever, the functionality will return to it. The adversary can arbitrarily delay the return and hence arbitrarily delay the rest of the execution of the calling protocol. However, there is no reason to allow the adversary to delay the execution of a locally executed algorithm such as SIGN_k, COMBINE or VERIFY. At the very least, the delay is a nuisance for the calling protocol, which needs to be designed to securely handle arbitrary interruptions in its execution. At worst, it prevents using the functionality as a subroutine in protocols that require uninterrupted execution to guarantee security or liveness/robustness (e.g., [CDL+25]). For these reasons, we design $\mathcal{F}$-TS3 such that all of its procedures return immediately and do not yield to the adversary.

Handling adversarially-controlled initialization. While $\mathcal{F}$-TS3[$\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}$] hardcodes several metadata values, such as the party set $\mathbf{P}$, thresholds $\mathbf{t}$, etc., $\mathcal{F}$-TS3 does not hardcode any inner workings of the algorithms KeyGen, Sign_k, Combine, and Verify, making the $\mathcal{F}$-TS3 functionality independent of any concrete threshold signature scheme. Instead, $\mathcal{F}$-TS3 asks the ideal adversary $\mathcal{S}$ to provide these algorithms, as well as public key pk and initial states $(\mathsf{st}_\mathcal{P}, \mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}$, during initialization. This is a standard approach for modeling ideal functionalities: The functionality does not care about the specific form of keys or signatures and hence lets the adversary decide. Unforgeability, correctness, and robustness will be enforced in Lines 25, 32 not matter how "bad" the algorithms output by $\mathcal{S}$ are. For example, if $\mathcal{S}$ outputs a verification algorithm Verify that blindly accepts all signatures (always outputs 1), the interface Verify will simply override its output whenever it violates unforgeability. Hence overall, protocols using $\mathcal{F}$-TS3 as a subroutine are assured that all properties hold, even if the adversary decides to output bad algorithms. Of course, when proving that a concrete protocol UC-realizes $\mathcal{F}$-TS3, the simulator $\mathcal{S}$ we construct for that proof will output the same secure algorithms that are used in the real world.

As observed in [CDL+25], a subtle issue is that during the initialization phase, $\mathcal{S}$ can potentially give control back to $\mathcal{Z}$, who may in turn instruct parties to start interacting with $\mathcal{F}$-TS3 while the details of the algorithms are still unspecified. One could naively prevent this by introducing a "ready" flag that is only set once the initialization is complete, and all other interfaces ignore calls until the flag is set. However, this renders $\mathcal{F}$-TS3 useless when used as a subprotocol since $\mathcal{S}$ can sabotage the initialization phase and thus prevent *all* further interaction with $\mathcal{F}$-TS3. We carefully handle this issue by setting default behaviors of Sign_k, Combine, and Verify as well as default values for pk and per-party states, which are used until $\mathcal{S}$ provides actual values. This way, parties can start interacting with $\mathcal{F}$-TS3 at any time while enjoying the guarantees of correctness, robustness, and unforgeability. If KeyGen is called by any party before $\mathcal{S}$ returns in Line 11, pk is set to its default value 1 cannot be overwritten by $\mathcal{S}$ anymore.

The concrete default values (Lines 6, 7, 8, and 14) we choose matter little. For example, the default Sign_k simply ignores its input and just sets the protocol message pm' and the states $\mathsf{st}'_\mathcal{P}, \mathsf{st}'_{\mathcal{P},\mathsf{eid}}$ to the arbitrary constant 1. As discussed above, any arbitrarily "bad" algorithm would do. Indeed, $\mathcal{S}$ is free to output these specific algorithms itself. Of course, the simulators $\mathcal{S}$ that we design are well-behaved and reply immediately to the request in Line 11, hence the default values are never used for our $\mathcal{S}$. They do however ensure that even if $\mathcal{S}$ is not well-behaved, the execution of $\mathcal{F}$-TS3 is not stalled.

Technically, Init is executed when the functionality is first created, which may coincide with the first command to it (see Section 2). This means that the first command to $\mathcal{F}$-TS3 *may* not be executed because Init yields to $\mathcal{S}$, which may delay, before the command is executed. We suggest that protocols establish an initialization phase, where all functionalities are created (and their Init procedures are run) before any actual commands are issued.

As also observed in [CDL+25], one challenge is to ensure that the algorithms Sign_k, Combine, and Verify output by $\mathcal{S}$ are indeed polynomial-time. We do not focus on this, but note that we can imagine executing each algorithm only for a certain number of steps and outputting some default value if it exceeds that limit.

Key generation. Key generation in $\mathcal{F}$-TS3 works as follows. During Init, the adversary $\mathcal{S}$ gets a chance to set pk (and $\mathsf{sk}_\mathcal{P}$) to any value it wants. If the adversary neglects to provide a public key before KeyGen is first called, KeyGen uses the arbitrary default $\mathsf{pk} = 1$. The environment (/calling protocol) needs to call $\mathcal{P}$.KeyGen() once before being able to instruct $\mathcal{P}$ to help create a signature via $\mathcal{P}$.Sign. The procedure $\mathcal{P}$.KeyGen immediately returns a public key pk, which all parties $\mathcal{P}' \in \mathbf{P}$ agree on.

Because KeyGen *immediately* returns pk and does not yield to the adversary, effectively the only way to realize KeyGen in the real world is via *trusted* key generation (see $\mathcal{F}$-Gen in Section 2). This is because in the ideal world, KeyGen returns immediately, so the real world protocol needs to return immediately as well and does not get time to execute, say, a distributed key generation protocol.[8] We make this design choice for simplicity, as our focus is on security for the signing procedure rather than key generation.

[8] Sending any network message as part of KeyGen in the real world would allow the environment (via the adversary) to observe this network message before receiving pk. In contrast, in the ideal world, pk is returned immediately. The simulator is not activated during KeyGen at all and hence cannot simulate the network message for the environment.

A simple way to modify $\mathcal{F}$-TS3 to allow for DKG-style key generation is by having $\mathcal{P}$.KeyGen activate the simulator $\mathcal{S}$ and ask for keys (instead of having $\mathcal{S}$ output $\mathsf{pk}, \mathsf{sk}_\mathcal{P}, \mathsf{st}_\mathcal{P}$ during Init). We use this modeling in $\mathcal{F}$-TS2 (Functionality 5), which can be easily ported to $\mathcal{F}$-TS3 (Ready[$\mathcal{P}$] would only be set once $\mathcal{S}$::KeyGen($\mathcal{P}$) returns). The downside of this modeling is as already discussed for signatures: yielding to the adversary allows the adversary to arbitrarily delay the process.

Another solution, closer to the spirit of $\mathcal{F}$-TS3, is to give key generation the same treatment as signing: Have $\mathrm{rnd}_{\mathrm{key}}$ many procedures KeyGen$_k$. Each KeyGen$_k$ takes the round $k-1$ DKG protocol messages as input, then computes and outputs the round k protocol message according to some algorithm KeyGen_k chosen by $\mathcal{S}$ during Init (or some default KeyGen_k if $\mathcal{S}$ has not output one yet). Like for signing, it is up to the environment to distribute these protocol messages to other parties. The last round $\mathcal{P}.\mathsf{KeyGen}_{\mathrm{rnd}_{\mathrm{key}}}$ would output the resulting public key pk (assuming it does not deviate from previously output pk for other parties) and mark party $\mathcal{P}$ to be ready to sign (Ready[$\mathcal{P}$] := 1). One would weaken $\mathsf{ValidateC}_\mathbf{t}, \mathsf{ValidateR}_\mathbf{t}, \mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ by making them check the messages sent and received during the key generation, too. For example, correctness (of signatures) is only guaranteed if key generation messages were correctly delivered.

Multiple instances and the $\mathsf{pk}' = \mathsf{pk}$ check. Following [Can04], our $\mathcal{F}$-TS3 functionality models unforgeability only for a single public key pk: In $\mathcal{V}$.Verify($\mathsf{pk}', \cdot, \cdot$) calls, $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ never outputs 1 if $\mathsf{pk}' \neq \mathsf{pk}$. This means that for all $\mathsf{pk}' \neq \mathsf{pk}$, calling $\mathcal{V}$.Verify($\mathsf{pk}', \cdot, \cdot$) is effectively useless (in that it does not offer any unforgeability guarantees at all).

In some sense, the lack of any guarantee for pk' is natural: of course the adversary/environment can trivially run the key generation procedure to obtain some public key $\mathsf{pk}' \neq \mathsf{pk}$ and create valid signatures for it. Still, it also means that if we have a protocol Π that generates multiple public keys, there needs to be one $\mathcal{F}$-TS3 instance for each public key, and in order to get useful Verify results, Π needs to pick the right $\mathcal{F}$-TS3 instance to call Verify on. Note that this is a nuisance rather than a strong limitation. It is a cost of abstraction: in the *real* world, one can safely call Verify on any instance of a threshold signature protocol instance Π — they will all yield the same result.

It is straightforward to modify $\mathcal{F}$-TS3 into a multi-instance version that supports multiple public keys with unforgeability guarantees. In particular, Sign$_k$ would take the targeted pk as additional input, and party set $\mathbf{P}$ and party state $\mathrm{ST}_\mathcal{P}, \mathsf{st}_\mathcal{P}$ would be kept separately for each pk. The cost of this modification is mostly notational complexity.

Corruption. Corruption works as described in Section 2. The adversary $\mathcal{S}$ can call $\mathcal{S}$.Corrupt($\mathcal{P}$) to corrupt $\mathcal{P}$. For static corruption (cor = static), this interface ignores the call if KeyGen has already been called for the $\mathcal{P}$ to be corrupted. Otherwise, the functionality adds $\mathcal{P}$ to the global set of corrupted parties $\mathbf{C}$ and returns the secret state associated with $\mathcal{P}$.

Corrupted parties forfeit their right for correctness, robustness, and unforgeability to be enforced. They are not allowed to call Verify or Combine. Additionally, for simplicity, corrupted parties are also not allowed to call Sign. However, of course, $\mathcal{S}$ can compute $\mathsf{Verify}, \mathsf{Combine}$ and Sign itself on behalf of corrupt parties.

Realizing $\mathcal{F}$-TS3. To realize $\mathcal{F}$-TS3, we consider a "wrapper protocol" Π-TS3 that executes the algorithms of a threshold signature scheme Σ_{TS} in the $\mathcal{F}$-Gen-hybrid model, where $\mathcal{F}$-Gen is a key generation functionality that provides parties with keys generated according to Σ_{TS}::KeyGen. Π-TS3 essentially specifies how Σ_{TS} should be executed in a UC setting, in such a way that their interfaces align with those of $\mathcal{F}$-TS3. To prove the equivalence of game-based security and UC security, we show that Π-TS3 UC-realizes $\mathcal{F}$-TS3 if and only if Σ_{TS} is correct, robust, and game-based unforgeable.

Theorem 2. *The protocol Π-TS3[$\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}, \mathsf{cor}$] (uniform-)UC-realizes $\mathcal{F}$-TS3[$\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}$] in the $\mathcal{F}$-Gen[$\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}$::Setup, Σ_{TS}::KeyGen] hybrid model in the presence of $\mathcal{G}$-Setup[Σ_{TS}::Setup] if and only if the threshold signature scheme $\Sigma_{\mathsf{TS}} = (\mathbf{rnd}, \mathsf{Setup}, \mathsf{KeyGen}, \{\mathsf{Sign}_k\}_{k=1}^{\mathrm{rnd}_{\mathrm{sign}}}, \mathsf{Verify}, \mathsf{ssid})$ is $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-correct (Definition 1), $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-robust (Definition 3), and $(\mathsf{cor}, \mathsf{lvl}, \mathbf{P}, \mathbf{t})$-unforgeable (Definition 4).*

The full proof is given in Appendix D. Here, we outline the main ideas. For the "if" direction, we construct a simulator $\mathcal{S}_{\mathsf{TS3}}$ that returns the algorithms specified by the threshold signature scheme Σ_{TS} to $\mathcal{F}$-TS3. Moreover, it simulates $\mathcal{F}$-Gen by internally running Σ_{TS}::KeyGen and provides the generated pk and initial per-party states to $\mathcal{F}$-TS3. To see why $\mathcal{S}_{\mathsf{TS3}}$ is a valid simulator, we note that the only gap between $\mathcal{F}$-TS3 and Π-TS3 lies in the Combine and Verify interfaces, as they

Protocol 1: Π-TS3$_{\mathcal{P}}[\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}, \mathsf{cor}]$ realizing $\mathcal{F}$-TS3

With access to $\mathcal{F}$-Gen$[\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}$::Setup, Σ_{TS}::KeyGen$]$.

INIT()
1: $\mathsf{pk} := \bot$
2: $\mathrm{ST}_{\mathcal{P}}[\mathsf{eid}] := (0, \bot, \ldots, \bot)$ for all eid
3: $\mathrm{Ready}[\mathcal{P}] := 0$
4: $\mathrm{SGN}[\cdot, \cdot] := \varnothing$ // Never read.

$\mathcal{P}$.KEYGEN() if $\mathcal{P} \in \mathbf{P}$
5: $(\mathsf{pk}, \mathsf{sk}_{\mathcal{P}}) \leftarrow \mathcal{F}$-Gen::$\mathcal{P}$.KEYGEN() // Trusted keygen.
6: $\mathsf{st}_{\mathcal{P}} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_{\mathcal{P}})$
7: $\mathrm{Ready}[\mathcal{P}] := 1$
8: **return** pk

$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$
9: $b :=_{\mathrm{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$ // Crucial difference to $\mathcal{F}$-TS3: no unforgeability check here.
10: **return** b

$\mathcal{P}$.SIGN$_k(\mathsf{eid}, \mathsf{inp})$, $k \in [\mathsf{rnd}_{\mathrm{sign}}]$ if $\mathcal{P} \in \mathbf{P}$
11: **assert** $\mathrm{Ready}[\mathcal{P}] = 1$
12: $\mathcal{O}_{\mathrm{SIGN},k}(\mathcal{P}, \mathsf{eid}, \mathsf{inp})$ // Verbatim

$\mathcal{P}$.COMBINE$(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [\mathsf{rnd}_{\mathrm{sign}}]})$
13: $\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [\mathsf{rnd}_{\mathrm{sign}}]})$
14: **return** σ

$\mathcal{A}$.CORRUPT()
15: **assert** $\mathcal{P} \in \mathbf{P}$
16: **assert** cor = adaptive $\vee$ [$\mathcal{P}$.KEYGEN has not been called yet]
17: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
18: **return** $(\mathsf{sk}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P}}, \mathrm{ST}_{\mathcal{P}})$

enforce the decision bits based on the predicates $\mathsf{ValidateC}_{\mathbf{t}}$, $\mathsf{ValidateR}_{\mathbf{t}}$, and $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$. However, if such enforcement happens with non-negligible probability, one can use $\mathcal{Z}$ to break the correctness, robustness or unforgeability of Σ_{TS}, contradicting our assumption. We emphasize that our simulator never gives control back to $\mathcal{Z}$ during initialization, as otherwise $\mathcal{Z}$ could instruct parties to interact with the interfaces with the default algorithms of $\mathcal{F}$-TS3. For the "only if" direction, we wrap any adversary $\mathcal{A}$ breaking correctness, robustness, or unforgeability into an environment $\mathcal{Z}$ distinguishing between the real and ideal world. Again, thanks to the alignment of interfaces, $\mathcal{Z}$ can perfectly emulate $\mathcal{A}$'s view in the real world by interacting with the parties running Π-TS3.

On UC and rewinding. One may question the result that game-based proofs can be translated into UC proofs given that many threshold signature schemes, such as Schnorr-based ones, rely on rewinding, which is generally considered to be incompatible with UC security. Indeed, in UC, the simulator $\mathcal{S}$ cannot rewind the environment $\mathcal{Z}$. However, this is not necessary in our setting: our simulator $\mathcal{S}_{\mathsf{TS3}}$ does not have any ambition to rewind anything. It simply outputs the keys and algorithms of Σ_{TS} to $\mathcal{F}$-TS3 and then becomes essentially inactive (besides some minor forwarding activity).

If we were to directly prove that, say, FROST (Protocol 9) UC-emulates $\mathcal{F}$-TS3, then this would ultimately involve a reduction algorithm $\mathcal{B}$ to some underlying hardness assumption. $\mathcal{B}$ runs the real-world UC experiment and, of course, is able to rewind that experiment, including the UC environment $\mathcal{Z}$, at will: the experiment is just another algorithm that $\mathcal{B}$ executes and interacts with.

Using $\mathcal{F}$-TS3. Because $\mathcal{F}$-TS3 leaves communication of protocol messages to the caller, it is relatively inconvenient to use (though not less convenient than a game-based definition). For this reason, we partially consider $\mathcal{F}$-TS3 a stepping stone for realizing more abstract functionalities such as $\mathcal{F}$-TS2, $\mathcal{F}$-TSSync2 (Sections 5 and 7) and perhaps future ideal threshold signature functionalities.

However, for specialized protocols that require fine-grained security and robustness guarantees, $\mathcal{F}$-TS3 may prove directly useful. For example, take $\mathcal{F}$-TS3 with $\mathsf{rnd}_{\mathrm{sign}} = 1$, i.e., non-interactive signing, which, for example, could be realized using BLS threshold signatures (using our protocol template in Appendix D). Additionally, consider a protocol that posts partial signatures to some public database until enough partial signatures are gathered and combined into one. For such a protocol, $\mathcal{F}$-TS3 is a good choice: SIGN$_1$ computes a partial signature, COMBINE allows combining them once enough partial signatures are collected. Helpfully, as discussed above, both SIGN$_1$ and COMBINE are modeled as local operations, i.e., the adversary does not get to delay their execution, which allows the protocol to consider generating and posting a partial signature as a single atomic step. Apart from that, our other functionalities (e.g., $\mathcal{F}$-TS2) do not expose partial signatures, so they would be unsuitable for such a scenario.

5 $\mathcal{F}$-TS2: Moderate Threshold Functionality

While $\mathcal{F}$-TS3 provides very general guarantees and aligns with the game-based syntax, its complexity hurts reusability. In particular, $\mathcal{F}$-TS3 leaves the scheduling of protocol messages to the outer pro-

Functionality 5: Moderate $\mathcal{F}$-TS2[$\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}$]

The code below references $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ from Game 3.

(⏲)Init()
1: $\mathsf{pk} := \bot$
2: $\mathrm{VER}[\cdot,\cdot,\cdot] := \bot$
3: $\mathrm{SGN}[\cdot,\cdot] := \varnothing$
4: $\mathrm{PP}_{\mathcal{P}}[\cdot] := \bot$ for all $\mathcal{P} \in \mathbf{P}$
5: $\mathsf{Verify} := (\cdot \mapsto 0)$ // Default
6: $\mathsf{Verify} \leftarrow$ ⏲$\mathcal{S}$::GetAlg()

(⏲)$\mathcal{P}$.KeyGen() for $\mathcal{P} \in \mathbf{P} \cup \{\mathcal{S}\}$
7: $\mathsf{pk}' \leftarrow$ ⏲$\mathcal{S}$::KeyGen($\mathcal{P}$)
8: **if** $\mathsf{pk} = \bot \wedge \forall m, \sigma :\ \mathrm{VER}[\mathsf{pk}', m, \sigma] \neq 1$ **then**
9: $\quad \mathsf{pk} := \mathsf{pk}'$
10: **return** pk

$\mathcal{S}$.Corrupt($\mathcal{P}$)
11: **assert** $\mathcal{P} \in \mathbf{P}$
12: **assert** cor = adaptive $\vee$ [$\mathcal{P}$.KeyGen has not been called yet]
13: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
14: **return** () // No secret to reveal.

(⏲)$\mathcal{P}$.Preprocess(eid) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
15: $\mathsf{pp} \leftarrow$ ⏲$\mathcal{S}$::Preprocess($\mathcal{P}, \mathsf{eid}$)
16: **if** $\mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] = \bot$ **then** $\mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] := \mathsf{pp}$
17: **return** pp

(⏲)$\mathcal{P}$.Sign($\boldsymbol{T}, m, \mathit{ctx}, (\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
18: **assert** $\mathcal{P} \in \boldsymbol{T}$
19: $\mathsf{eid} := \mathsf{eid}_{\mathcal{P}}$
20: **assert** $\mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] = \mathsf{pp}_{\mathcal{P}} \neq \bot$
21: $\mathit{ssid} := \mathsf{ssid}(\boldsymbol{T}, m, \mathit{ctx}, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$
22: $\mathrm{SGN}[m, \mathit{ssid}] := \mathrm{SGN}[m, \mathit{ssid}] \cup \{\mathcal{P}\}$
23: $\sigma \leftarrow$ ⏲$\mathcal{S}$::Sign($\mathcal{P}, \boldsymbol{T}, m, \mathit{ctx}, (\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$)
24: **assert** $\mathcal{P}$.Verify(pk, m, σ) $= 1$
25: **return** σ

$\mathcal{V}$.Verify(pk', m, σ) for any honest party $\mathcal{V} \notin \mathbf{C}$
26: $b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
27: **if** $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}', m)$ **then**
28: $\quad b := 0$ // Unforgeability
29: **if** $\mathrm{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
30: $\quad \mathrm{VER}[\mathsf{pk}', m, \sigma] := b$
31: **return** $\mathrm{VER}[\mathsf{pk}', m, \sigma]$

tocol, which goes against the modular design of UC-secure protocols. In this section, we present a simpler functionality, $\mathcal{F}$-TS2 (Functionality 5) that still captures most relevant security guarantees but provides a simpler interface that is easier to use.

$\mathcal{F}$-TS2 is similar to $\mathcal{F}$-TS3 but makes the following simplifications: (1) it assumes a simple preprocessing phase that is independent of $(\boldsymbol{T}, m, \mathit{ctx})$, captured by the Preprocess interface, and (2) it consolidates the (online) signing and combining phases into a single Sign interface, assuming that *ssid* can be determined at the start of signing. The advantage of this approach is that the outer protocol does not need to manage multiple rounds of message exchanges, but can simply call Sign after collecting preprocessed protocol messages $(\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$ and receive a signature in response. At the same time, $\mathcal{F}$-TS2 is specifically designed to meaningfully model all levels of unforgeability lvl and most popular choices of ssid. In particular, this includes ssid that depends on preprocessing values $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$, such as FROST. The only restriction is that $\mathcal{F}$-TS2 syntactically supports at most a single round of preprocessing messages to be input to ssid (see Line 21 of $\mathcal{F}$-TS2). Thankfully, this covers most important schemes.

The cleaner design of $\mathcal{F}$-TS2 comes at the cost of expressiveness. Since we aim to capture protocols that make no assumptions about the communication channels (as is common in most threshold schemes in practice), the main limitation of $\mathcal{F}$-TS2 is that it cannot model correctness and robustness. This is inevitable because, without any assumptions about the communication channels, the adversary can always block messages between honest parties, causing honest parties to fail to sign or to output an invalid signature.[9] The only notion of correctness that $\mathcal{F}$-TS2 captures is that whenever Sign outputs a non-$\bot$ signature, then it is valid (trivially enforced in Line 24 of Functionality 5). This is a standard notion of correctness for MPC protocols, but makes no guarantees that Sign *ever* outputs any non-$\bot$ value. Correctness in a game-based sense is much stronger: if all parties are honest and protocol messages are delivered correctly, then the produced signature is valid. Due to the limitations with correctness, we focus on capturing unforgeability with $\mathcal{F}$-TS2.

The interface of $\mathcal{F}$-TS2. We give an overview of how $\mathcal{F}$-TS2 works and how to use it. When $\mathcal{F}$-TS2 is first initialized, it asks the simulator $\mathcal{S}$ for the verification algorithm Verify. Similarly to $\mathcal{F}$-TS3, until $\mathcal{S}$ responds, we use a default verification algorithm that always returns 0 (i.e., all signatures are invalid). This means that all operations of $\mathcal{F}$-TS2 are well-defined even if $\mathcal{S}$ is late to respond.

[9] We address these issues in Section 7 by assuming synchronous communication.

We expect the environment to call $\mathcal{P}$.KeyGen() for all signers $\mathcal{P} \in \mathbf{P}$. As indicated by the ($\clubsuit$) symbol, $\mathcal{P}$.KeyGen() does not return immediately. Instead, it allows the simulator to delay key generation arbitrarily.

Once $\mathcal{P}$.KeyGen() returns a public key pk, we imagine that $\mathcal{P}$ has received their secret key. At that point, the environment may call $\mathcal{P}$.Preprocess($\mathsf{eid}_\mathcal{P}$) for some arbitrary execution ID $\mathsf{eid}_\mathcal{P}$ to retrieve a preprocessing value $\mathsf{pp}_\mathcal{P}$. Note that this preprocessing phase is independent of the signing set $\boldsymbol{T}$, the message m, and the context ctx. It is then the environment's responsibility to distribute $\mathsf{eid}_\mathcal{P}, \mathsf{pp}_\mathcal{P}$ to the (other) intended signers $\mathcal{Q} \in \boldsymbol{T}$.

When it comes to correctness, for a signature to be created, we expect all signers $\mathcal{P} \in \boldsymbol{T}$ to agree on the same signing data $(\boldsymbol{T}, m, ctx, (\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}})$ at the start of signing. Additionally, each $\mathsf{eid}_\mathcal{P}$ must only be used for a single signing process (instead of, say, trying to sign two different messages involving the same $\mathsf{eid}_\mathcal{P}$). Strictly speaking, these requirements are not captured in $\mathcal{F}$-TS2, which does not provide any guarantees for valid signatures to be created *at all*. However, for our protocol Π-TS2 (just like for $\mathcal{F}$-TS3 and our game-based defintion), this is the expectation.

Once a signer is aware of signing data $(\boldsymbol{T}, m, ctx, (\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}})$, it can compute the signing session ID $ssid := \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}})$. By computing $ssid$, the environment can make the decision whether to agree to creating a signature for $m, ssid$. As discussed in Section 3.4 and Table 3, for sufficiently high security levels lvl, unforgeability guarantees that a signature for m will only created if all signers agree on $ssid$. In the case of strong unforgeability, agreeing to sign m with $ssid'$ after already having signed with $ssid \neq ssid'$ will also mean agreeing to creating a second (different) signature on m. In both cases, computing and inspecting $ssid$ helps the environment make an informed decision about whether to agree to signing or not. If the environment is okay with a signature being created, it calls $\mathcal{P}$.Sign($\boldsymbol{T}, m, ctx, (\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$), which may eventually return a signature σ. There are no guarantees that all parties will agree on the signature σ, nor that any party will ever receive a non-$\perp$ signature.

The environment can call $\mathcal{V}$.Verify(pk', m, σ) for any party $\mathcal{V}$ (in particular $\mathcal{V} \notin \mathbf{P}$) to check the validity of σ on m for any public key pk'. In contrast to all other $\mathcal{F}$-TS2 operations, Verify is guaranteed to immediately return a verification bit without yielding to the adversary $\mathcal{S}$. As discussed in Section 4, Verify does not provide any unforgeability guarantees for $\mathsf{pk}' \neq \mathsf{pk}$, i.e., the signature verification for any $\mathcal{F}$-TS2 instance is only good for a single public key pk.

The execution ID eid. We have used the (local) execution ID eid before in $\mathcal{F}$-TS3 and in the game-based definition. eid is an arbitrary bit string that uniquely identifies a signing process. In $\mathcal{F}$-TS3 and the game, eid is a pure a modeling artifact, allowing the environment/adversary to point to which execution's state the functionality/oracle should use. None of the threshold signature scheme algorithms receive eid as input.

In $\mathcal{F}$-TS2, the execution ID eid takes a slightly more active role: As described above, signers shall agree on the same $(\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$. This is for the benefit of implemeting protocols, such as Π-TS2, that use $\mathsf{eid}_\mathcal{Q}$ as routing information: When $\mathcal{P}$ sends a protocol message pm to $\mathcal{Q}$ during the signing process, it specifies $\mathsf{eid}_\mathcal{Q}$ as the port of the message, so that $\mathcal{Q}$ can correctly assign pm to the right signing process.

In practice and in the literature, this routing function of $\mathsf{eid}_\mathcal{Q}$ is often replaced by the preprocessing values $\mathsf{pp}_\mathcal{Q}$. This is perfectly reasonable, assuming that $\mathsf{pp}_\mathcal{Q}$ has enough entropy to uniquely identify the signing process. However, $\mathcal{F}$-TS2 does not enforce any entropy requirements on pp: the UC simulator chooses pp (and even in the game-based definition, there is no requirement for pp to have high entropy). For this reason, we make eid an explicit part of the input.

Modeling preprocessing. One may wonder why we model preprocessing in a setting where signing protocols are allowed to take arbitrarily many rounds. What is the benefit of preprocessing if $\mathcal{F}$-TS2 does not enforce that anything meaningful happens during Preprocess? First, it is indeed that case that preprocessing is functionally optional in $\mathcal{F}$-TS2. Indeed, our protocol Π-TS2 skips preprocessing entirely (setting $\mathsf{pp} := \perp$) if its underlying $\mathcal{F}$-TS3 does not support preprocessing. Second, implementing protocols may run meaningful preprocessing, and modeling preprocessing in $\mathcal{F}$-TS2 allows the calling protocol to support this syntactically. Third, and perhaps most importantly, the unforgeability notion of many important schemes (e.g., UF-2 FROST [BCK+22, BTZ22]) is strongly tied to their preprocessing via ssid, and modeling preprocessing allows $\mathcal{F}$-TS2 to model those notions faithfully. Concretely, $\mathcal{F}$-TS2 makes the preprocessing messages an explicit and caller-accessible arti-

fact (whereas other rounds' messages are hidden within the model). This enables $\mathcal{F}$-TS2 to feed the preprocessing values into ssid and make the unforgeability notion depend on it.

With the motivation out of the way, we turn to discussing modeling details of preprocessing. The environment may call $\mathcal{P}$.Preprocess($\mathsf{eid}_\mathcal{P}$) for some arbitrary execution ID $\mathsf{eid}_\mathcal{P}$ to retrieve a preprocessing value $\mathsf{pp}_\mathcal{P}$. One should imagine that this happens in anticipation of future signing processes, allowing parties to execute some computations in advance (say, when idle over night).

It is the environment's responsibility to distribute $\mathsf{eid}_\mathcal{P}, \mathsf{pp}_\mathcal{P}$ to the (other) intended signers $\mathcal{Q} \in \boldsymbol{T}$. There are two main ways of implementing this distribution, which $\mathcal{F}$-TS2 is agnostic about:

- **Peer to peer.** $\mathcal{P}$ sends $\mathsf{eid}_\mathcal{P}, \mathsf{pp}_\mathcal{P}$ to each $\mathcal{Q} \in \boldsymbol{T}$ directly at preprocessing time. At signing time, the parties simply input the values $(\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$ received earlier into Sign. Some coordination is needed to ensure that all parties agree on the same $(\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$ at signing time.
- **Echo scheme.** $\mathcal{P}$ sends $\mathsf{eid}_\mathcal{P}, \mathsf{pp}_\mathcal{P}$ to some "leader" party $\mathcal{L}$, who collects several of these values from all parties. At signing time, $\mathcal{L}$ sends a *leader request* to all $\mathcal{Q} \in \boldsymbol{T}$, containing the signing set $\boldsymbol{T}$, the message to be signed m, the context ctx, and any list of prior received preprocessing values $(\mathsf{eid}_\mathcal{Q}, \mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$. No additional coordination is needed, the leader request itself ensures that all parties agree on the same values.

The peer to peer approach causes $|\boldsymbol{T}|^2$ messages each containing 1 preprocessing value to be sent at preprocessing time, whereas the echo scheme approach [BTZ22, BCK+22] causes $|\boldsymbol{T}|$ messages with 1 preprocessing value each at preprocessing time, and $|\boldsymbol{T}|$ messages with $|\boldsymbol{T}|$ preprocessing values each at signing time.

In principle, $\mathcal{F}$-TS2 allows preprocessing to be implemented by an interactive protocol, but our instantiation Π-TS2 implements it as a non-interactive protocol, where the signer simply executes the local operation Sign_1.

To model protocols without preprocessing in the UF-0 or UF-1 security levels, we recommend using $\mathcal{F}$-TS1 in Section 6. Security levels (S)UF-3, (S)UF-4 effectively require preprocessing to be meaningful. To model protocols without preprocessing for level (S)UF-2, one may modify $\mathcal{F}$-TS2 as follows. First, remove the Preprocess interface and the PP data structure. From Sign, remove the $\mathsf{pp}_\mathcal{Q}$ inputs and the assertion about PP and $\mathsf{pp}_\mathcal{P}$ in Line 20. Remove $\mathsf{pp}_\mathcal{Q}$ from the input of ssid in Line 21 and from the input of $\mathcal{S}$::Sign in Line 23. Note that this restricts what ssid can be modeled with this modified $\mathcal{F}$-TS2.

Key generation. The only guarantees enforced by KeyGen in $\mathcal{F}$-TS2 are *agreement* (Line 8 and 10) and *no pre-forgeries* (Line 8). Agreement means that no two honest parties return different public keys. No pre-forgeries is modeled by requiring that the output public key pk' be "fresh", i.e., there must not be a valid signature on any message under pk' (formally, we require that $\forall m, \sigma :\ \mathrm{VER}[\mathsf{pk}', m, \sigma] \neq 1$). This avoids the undesirable situation where some signature σ is declared valid for pk', m before pk' is output by KeyGen and hence before $\mathcal{F}$-TS2 starts enforcing unforgeability for pk'. Consider the following concrete example: $\mathcal{S}$ outputs $\mathsf{Verify} = (\cdot \mapsto 1)$ (e.g., all signatures are valid); the environment calls $\mathcal{V}$.Verify($\mathsf{pk}', m = 0, \sigma = 0$), which declares $\sigma = 0$ to be a valid signature on $m = 0$ for public key pk' (note that still $\mathsf{pk} = \bot$ at this point, so the unforgeability check in Line 28 does not apply); then the environment calls $\mathcal{P}$.KeyGen and $\mathcal{S}$ sets $\mathsf{pk} = \mathsf{pk}'$. In that example, the parties are now using a public key $\mathsf{pk} = \mathsf{pk}'$ for which there is already a "pre-forgery" σ out there. Clearly, this is undesirable and from a good DKG, we would expect that pk' is fresh and does not correspond to any prior key for which signatures may already have been created. The no-preforgery check $\forall m, \sigma :\ \mathrm{VER}[\mathsf{pk}', m, \sigma] \neq 1$ in Line 8 ensures that $\mathcal{S}$ cannot output a public key pk' for which there are pre-forgeries. Note that this check is not needed in $\mathcal{F}$-TS3 because there, before pk is set, all signatures are declared invalid by the default Verify algorithm.

In contrast to $\mathcal{F}$-TS3, the functionality $\mathcal{F}$-TS2 allows the adversary to delay or block key generation. This is in line with $\mathcal{F}$-TS2 not being concerned with correctness or robustness. A positive side effect of this is that the key generation part of $\mathcal{F}$-TS2 can be reasonably implemented with a distributed key generation protocol (DKG). This is in contrast to $\mathcal{F}$-TS3, which effectively requires trusted key generation (see discussion in Section 4). We note that because our protocol Π-TS2 uses $\mathcal{F}$-TS3 as a hybrid, Π-TS2 also ultimately relies on trusted key generation. Other instantiations of $\mathcal{F}$-TS2 may use a DKG.

Note that $\mathcal{F}$-TS2 explicitly allows the simulator $\mathcal{S}$ to call $\mathcal{S}$.KeyGen. This simply models that $\mathcal{S}$ can set pk (once) at any time, even before the environment calls $\mathcal{P}$.KeyGen. This is not a detriment to security: It merely allows $\mathcal{S}$ to make $\mathcal{F}$-TS2 start enforcing unforgeability "early". This is a necessary

proof detail for our realization Π-TS2, which is based on $\mathcal{F}$-TS3 and starts enforcing unforgeability from the moment Verify is set.

Realizing $\mathcal{F}$-TS2. We now show the existence of a protocol Π-TS2 that UC-realizes $\mathcal{F}$-TS2 in the $\mathcal{F}$-TS3 hybrid model. At a high level, Π-TS2 uses $\mathcal{F}$-TS3 to implement PREPROCESS by querying $\mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_1$ and obtaining offline protocol messages. It then implements SIGN by querying $\mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_k$ for $k \geq 2$ with the collected offline messages, and $\mathcal{F}$-TS3::$\mathcal{P}$.COMBINE returns the resulting signature. To exchange protocol messages, Π-TS2 uses an insecure channel functionality $\mathcal{F}$-Net, which can be entirely controlled by the adversary, to model the lack of communication assumptions. While we do not guarantee correctness, it is easy to see that Π-TS2 always produces a valid signature when $\boldsymbol{T} \subseteq \mathbf{H}$ and $(\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$ are generated via previous invocations of PREPROCESS, assuming benign adversaries that deliver messages between honest parties through $\mathcal{F}$-Net without tampering or delay. The proof of the following theorem is given in Appendix E. At a high level, the simulator $\mathcal{S}_{\text{TS2}}$ internally runs a copy of the real protocol Π-TS2 whenever $\mathcal{F}$-TS2 requests it to perform KEYGEN, PREPROCESS, or SIGN on behalf of a (dummy) honest party.

Theorem 3. *Let* $\mathsf{rnd}_{\text{com}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{ctx}} \in \{1, 2\}$. *Let* $t_{\text{correct}} = t_{\text{robust}} = |\mathbf{P}| + 1$ *(i.e., no correctness or robustness guarantees in* $\mathcal{F}$-TS3*). The protocol* Π-TS2$[\mathbf{P}, \mathbf{rnd}, \mathsf{cor}]$ *(uniform-)UC-realizes* $\mathcal{F}$-TS2$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}]$ *in the* $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ *and* $\mathcal{F}$-Net *hybrid model.*

The protocol Π-TS2 supports two modes: $\mathsf{rnd}_{\text{com}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{ctx}} = 2$, which is the preprocessing mode, and $\mathsf{rnd}_{\text{com}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{ctx}} = 1$, which is a mode for protocols without preprocessing. In the latter case, Π-TS2 simply outputs empty preprocessing $\mathsf{pp} := \bot$ and ssid in $\mathcal{F}$-TS2 ignores the preprocessing inputs.

Using $\mathcal{F}$-TS2. $\mathcal{F}$-TS2 is well-suited to be used as a subroutine in a larger protocol that already follows the (widespread) default MPC-correctness–style without guaranteed output delivery. This MPC-correctness–style means that the adversary (in the larger protocol) is already allowed to delay honest parties arbitrarily, or make them output $\bot$, but cannot make honest parties output a wrong result. $\mathcal{F}$-TS2 follows the same paradigm: the UC adversary can delay honest parties' key generation, preprocessing, and signing processes arbitrarily, and can make them fail, outputting $\bot$, but cannot cause them to output an invalid signature. Stronger correctness (or perhaps even robustness guarantees) can be argued for the concrete protocol outside of UC. In this setting, UC is "only" used to argue about composable security, not correctness.

$\mathcal{F}$-TS2 is much easier to use compared to $\mathcal{F}$-TS3: The $\mathcal{F}$-TS2 caller is not tasked with delivering threshold signature protocol messages between the signers. Instead, it is the implementing protocol's responsibility to communicate protocol messages internally (see Π-TS2 (Protocol 5)).

A lot of the modeling of $\mathcal{F}$-TS2 serves supporting different lvl and ssid choices by making preprocessing explicit. In Section 6, we further simplify $\mathcal{F}$-TS2 by restricting the security notion to $\mathsf{lvl} =$ UF-1 (or $\mathsf{lvl} =$ UF-0), which is agnostic of ssid and preprocessing.

6 $\mathcal{F}$-TS1: Simplest Threshold Functionality

We define the simplest threshold signature functionality $\mathcal{F}$-TS1 (Functionality 6). Essentially, we trade the granularity of $\mathcal{F}$-TS2 for simplicity, making it even easier to use in protocols if higher unforgeability levels are not a concern. Compared to $\mathcal{F}$-TS2, it only has a single interface for signing without preprocessing, and it does not keep track of *ssid*. As such, the VERIFY interface hardcodes the unforgeability level UF-1 (i.e., a signature is considered trivial to produce if at least $t_{\text{UF}} - |\mathbf{C}|$ honest parties have previously signed the same message). One can easily modify the functionality to model UF-0 by changing the $|\text{SGN}[m] \cup \mathbf{C}| < t_{\text{UF}}$ condition in VERIFY to $\text{SGN}[m] \cap \mathbf{H} = \varnothing$.

Realizing $\mathcal{F}$-TS1. We provide a protocol Π-TS1 (Protocol 6) that UC-realizes $\mathcal{F}$-TS1 in the $\mathcal{F}$-TS3 and $\mathcal{F}$-Net hybrid model. Π-TS1 essentially uses $\mathcal{F}$-TS3 to generate protocol messages and exchanges them via $\mathcal{F}$-Net The proof of the following theorem is given in Appendix F. Similar to Theorem 3, the proof constructs a simulator $\mathcal{S}_{\text{TS1}}$ that internally runs a copy of the adversary $\mathcal{A}$ and essentially simulates Π-TS1 by running it verbatim: there are no secrets kept from $\mathcal{S}_{\text{TS1}}$ by $\mathcal{F}$-TS1.

Theorem 4. *Let* $\mathsf{lvl} =$ UF-1. *Let* $\mathsf{rnd}_{\text{com}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{ctx}} = 1$. *Let* $t_{\text{correct}} = t_{\text{robust}} = |\mathbf{P}|+1$ *(i.e., no correctness or robustness guarantees in* $\mathcal{F}$-TS3*). The protocol* Π-TS1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ *(uniform-)UC-realizes* $\mathcal{F}$-TS1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ *in the* $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ *and* $\mathcal{F}$-Net *hybrid model.*

Functionality 6: Simple $\mathcal{F}$-TS1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$

$(\clubsuit)$INIT()
1: $\mathsf{pk} := \bot$
2: $\mathrm{VER}[\cdot,\cdot,\cdot] := \bot$
3: $\mathrm{SGN}[\cdot] := \varnothing$
4: $\mathsf{Verify} := (\cdot \mapsto 0)$ // Default
5: $\mathsf{Verify} \leftarrow \clubsuit\mathcal{S}$::GETALG()

$(\clubsuit)\mathcal{P}$.KEYGEN() for $\mathcal{P} \in \mathbf{P} \cup \{\mathcal{S}\}$
6: $\mathsf{pk}' \leftarrow \clubsuit\mathcal{S}$::KEYGEN$(\mathcal{P})$
7: **if** $\mathsf{pk} = \bot \wedge \forall m, \sigma :\ \mathrm{VER}[\mathsf{pk}', m, \sigma] \neq 1$ **then**
8: $\quad \mathsf{pk} := \mathsf{pk}'$
9: **return** pk

$\mathcal{S}$.CORRUPT$(\mathcal{P})$
10: **assert** $\mathcal{P} \in \mathbf{P}$
11: **assert** cor = adaptive $\vee$ [$\mathcal{P}$.KEYGEN has not been called yet]
12: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
13: **return** () // No secret to reveal.

$(\clubsuit)\mathcal{P}$.SIGN$(\boldsymbol{T}, m)$ for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
14: **assert** $\mathcal{P} \in \boldsymbol{T}$
15: $\mathrm{SGN}[m] := \mathrm{SGN}[m] \cup \{\mathcal{P}\}$
16: $\sigma \leftarrow \clubsuit\mathcal{S}$::SIGN$(\mathcal{P}, \boldsymbol{T}, m)$
17: **assert** $\mathcal{P}$.VERIFY$(\mathsf{pk}, m, \sigma) = 1$
18: **return** σ

$\mathcal{V}$.VERIFY$(\mathsf{pk}', m, \sigma)$ for any honest party $\mathcal{V} \notin \mathbf{C}$
19: $b :=_{\mathrm{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
20: **if** $\mathsf{pk}' = \mathsf{pk} \wedge |\mathbf{C}| < t_{\mathrm{corrupt}}$
21: $\wedge |\mathrm{SGN}[m] \cup \mathbf{C}| < t_{\mathrm{UF}}$ // UF-1 **then**
22: $\quad b := 0$
23: **if** $\mathrm{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
24: $\quad \mathrm{VER}[\mathsf{pk}', m, \sigma] := b$
25: **return** $\mathrm{VER}[\mathsf{pk}', m, \sigma]$

7 A Moderate UC Functionality for the Synchronous Setting

Motivated by $\mathcal{F}$-TS2 not providing robustness (or correctness) guarantees, we present a new threshold signature functionality $\mathcal{F}$-TSSync2 that provides these guarantees in a synchronous setting. First, let us reflect on the reasons why $\mathcal{F}$-TS2 does not provide meaningful correctness or robustness guarantees. The main issue is that the fuctionality, upon being asked to sign a message, will ask the simulator for a signature. This yields control to the simulator, which then chooses at what point a signature is being returned. In our concrete protocol Π-TS2, this happens once the real world adversary has decided to deliver the network messages for all rounds of the protocol. In this sense, asking the simulator is good modeling for a threshold signature ideal functionality: it captures the fact that a signature will not immediately materialize, but it will take some time (during which other events happen, like concurrent protocols or corruptions).

However, it leaves open the possibility that the simulator never returns a signature at all. For both robustness and correctness, we would expect that (at least under certain circumstances), a signature will be returned after some bounded amount of time. There is no straightforward way for an ideal functionality to express that even though the simulator is supposed to delay, it must only delay by some bounded amount of time. It is all (simulator gets to delay, meanning also delay indefinitely) or nothing (signature gets returned immediately). One may, of course, prove that some real UC protocol has some correctness and robustness properties by defining these notions as properties of a protocol rather than modeling these properties with an ideal functionality. Alternatively, one may use our $\mathcal{F}$-TS3 functionality, which does provide correctness and robustness guarantees. However, $\mathcal{F}$-TS3 exposes the threshold signature scheme's round function and leaves it up to the environment to deliver protocol messages pm. This makes $\mathcal{F}$-TS3 difficult to use in larger protocols, which need to handle protocol message delivery. Overall, it is clearly preferable to have robustness and correctness properties guaranteed by the ideal functionality itself (unlike $\mathcal{F}$-TS2) while still hiding protocol execution details (unlike $\mathcal{F}$-TS3), as both aspects make it easier to use the functionality as a subprotocol in larger protocols.

A solution in the synchronous setting using (global) clocks. We utilize a synchronous setting, where time progresses in *rounds*. Essentially, we imagine that each round, every (honest) party is invoked, reads messages sent in the previous round, executes some computation, and sends messages to be received in the next round. One crucial ingredient for the synchronous setting is a mechanism to ensure that before we progress to the next round, all honest parties must have completed their actions for the current round. This is achieved by an ideal clock functionality $\mathcal{G}$-Clock (see Functionality 7), which keeps track of which parties (and which of their protocol sessions *sid*) have finished executing their round (which they indicate by calling ROUNDOK) and increments the round counter *now* once

Functionality 7: $\mathcal{G}$-Clock

An ideal clock functionality. Used to abstract universal agreement on whether all honest parties are done executing their round. In practice, this agreement could be implemented using loosely synchronized clocks (and an upper-bound on the network delay).

Init() // Invoked only upon creation
1: Done$[\mathcal{P}, sid] := 0$ for all $(\mathcal{P}, sid)$
2: Reg $:= \varnothing$ // Set of registered $(\mathcal{P}, sid)$
3: $now := 1$ // Global round counter

$\mathcal{P}$.GetRound() for all $\mathcal{P}$ from all sid // Poll after calling RoundOK.
4: // Increment round if all honest parties in all sessions are done.
5: **if** $\forall(\mathcal{Q}, sid') \in$ Reg $: \mathcal{Q} \in \mathbf{H} \Rightarrow$ Done$[\mathcal{Q}, sid'] = 1$ **then**
6: Done$[\mathcal{Q}, sid'] := 0$ for all $(\mathcal{Q}, sid') \in$ Reg
7: $now := now + 1$
8: **return** now

$\mathcal{P}$.Register() for all $\mathcal{P}$ from all sid // Party $\mathcal{P}$ in session sid registers to use the clock
9: Let sid be the caller's session ID.
10: Reg $:=$ Reg $\cup \{(\mathcal{P}, sid)\}$

($\circlearrowright$)$\mathcal{P}$.RoundOK() for all $\mathcal{P}$ from all sid // Party $\mathcal{P}$ in session sid is done with its round
11: Let sid be the caller's session ID.
12: $\mathcal{P}$.Register() // Ensure registered
13: Done$[\mathcal{P}, sid] := 1$
14: $\mathcal{P}$.GetRound() // Update round if possible
15: $\circlearrowright\mathcal{A}$::RoundOK$(\mathcal{P}, sid)$ // Notify $\mathcal{A}$ (enable corrupted parties to act every round)

all honest parties (in all their protocol sessions) have done so. The round counter now is available to all parties via GetRound().

This modeling is heavily inspired by [KMTZ13]. One difference is that we define a *global* clock functionality. This means the same clock is used across multiple protocols and sessions, which we believe is the right modeling for compositions of multiple protocols in a synchronous setting. It also allows us to reference $\mathcal{G}$-Clock from within $\mathcal{F}$-TSSync2, which simplifies the modeling. In our $\mathcal{F}$-TSSync2 setting, we will generally think of one $\mathcal{F}$-TS2 instance with session ID sid_{tsig}, while all other protocols and sessions are modeled by the environment, which can play the role of arbitrary parties $\mathcal{P}$ in arbitrary sessions $sid \neq sid_{\text{tsig}}$. Time only progresses if for all registered $(\mathcal{P}, sid)$, the RoundOK procedure was called, i.e., both $\mathcal{F}$-TSSync2 and concurrently running protocols have finished their round.

We also adapt the original modeling to handle adaptive corruption. The [KMTZ13] functionality gets stuck in the same round forever if the last honest party missing to complete the round gets adaptively corrupted and never calls RoundOK(). In our $\mathcal{G}$-Clock, calling GetRound() will increment the round in these cases. Additionally, we model an authenticated synchronous network via an ideal functionality $\mathcal{F}$-NetSync (see Functionality 8), which queries $\mathcal{G}$-Clock to determine the current round. Messages are available one round after they were sent. This network will be used in the real world protocol Π-TSSync2 (see Protocol 7).

Functionality 8: $\mathcal{F}$-NetSync[$\mathbf{P}$]

An authenticated synchronous (round-based) group communication channel. Has access to $\mathcal{G}$-Clock.

Init() // Invoked only upon creation
1: Inbox$[\mathcal{P}] := \perp$ for all $\mathcal{P} \in \mathbf{P}$
2: Outbox$[round, \mathcal{P}, \mathcal{Q}, port] := \perp$ for all $\mathcal{P}, \mathcal{Q} \in \mathbf{P}, round \in \mathbb{N}, port \in \{0,1\}^*$

$\mathcal{P}$.Send$(\mathcal{Q}, port, \mathsf{pm})$ for all $\mathcal{P} \in \mathbf{P}$ // Send pm to party $\mathcal{Q}$ on port $port$
3: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
4: **assert** Outbox$[now, \mathcal{P}, \mathcal{Q}, port] = \perp$ // $\mathcal{P}$ has not yet sent this round to party $\mathcal{Q}$ on this port
5: Outbox$[now, \mathcal{P}, \mathcal{Q}, port] :=$ pm

$\mathcal{P}$.Receive$(port)$ for all $\mathcal{P} \in \mathbf{P}$ // Retrieve messages sent to $\mathcal{P}$ in previous round on port $port$
6: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
7: Inbox $:=$ (Outbox$[now - 1, \mathcal{Q}, \mathcal{P}, port])_{\mathcal{Q} \in \mathbf{P}}$ // Retrieve messages addressed to $\mathcal{P}, port$ last round
8: **return** Inbox

$\mathcal{A}$.GetOutbox() // Leak sent messages to adversary
9: **return** Outbox

How $\mathcal{F}$-TSSync2 works. With this machinery in place, we can now define our new threshold signature functionality (Functionality 9). $\mathcal{F}$-TSSync2 models trusted key generation as in $\mathcal{F}$-TS3. To create a signature, one needs to execute preprocessing (which is a local operation), then call Sign for each signer to indicate the intention to create a signature. This queues the signing process to start in the current round (if $\mathcal{P}$ has not yet called Tick() yet) or the next round (if $\mathcal{P}$ has called Tick() this round already). Every round, the environment must call $\mathcal{P}$.Tick() to execute the round for $\mathcal{P}$. This will progress time after all honest parties (in this protocol and other concurrent ones) have executed their round. After $\mathsf{rnd}_{\text{sign}} - 1$ rounds[10] have passed, the signature can be retrieved by calling GetSig. Usually, the signature returned there will have been previously set by the simulator via $\mathcal{S}$.SetSig (if the simulator is trying to emulate some real protocol). If the simulator does not set a signature (or one that is not valid), but enough signers were participating such that we would expect correctness or robustness guarantees, GetSig use ForceValidity to ensure a valid signature is output just like $\mathcal{F}$-TS3. This models correctness and robustness: the functionality guarantees that qualified signing processes *will* result in a valid signature after $\mathsf{rnd}_{\text{sign}}$ rounds. Unforgeability is enforced via Verify as in the other functionalities.

Guaranteeing liveness. A core design goal of $\mathcal{F}$-TSSync2 is that the ideal world adversary (the simulator) must not be able to prevent the environment (which represents the protocol using $\mathcal{F}$-TSSync2 as a subprotocol) from making progress (towards creating a signature). We achieve this by not yielding control to the ideal world adversary in any procedure except for Init (to ask the simulator for algorithms) and Tick (to ensure corrupted parties are allowed to act every round and to allow the simulator to simulate network messages of honest parties). The procedures that may not immediately return (due to yielding to the ideal world adversary) are indicated with the (🕒) symbol in Functionality 9. This is the crucial difference to $\mathcal{F}$-TS2, where invoking, say, Sign would yield control to the simulator, meaning that a prompt result is not guaranteed. To manage the balancing act of allowing the simulator to properly simulate the protocol and interact with the functionality, but not allowing it to prevent progress, we make the following modeling decisions. (1) The simulator outputs a preprocessing algorithm Preprocess (if it does not do so in time, we will use a default algorithm that always outputs $\mathsf{pp} = 1$). This allows $\mathcal{F}$-TSSync2 to reply with preprocessing values immediately upon request, without needing to yield to the simulator at that point. To model potentially stateful preprocessing as in $\mathcal{F}$-TS3 and Section 3, Preprocess gets a state variable st_{pp} that it can read and update, and the simulator may also read and update st_{pp} (via $\mathcal{S}$.GetProcessInfo and $\mathcal{S}$.SetPreprocState, respectively). (2) For the simulator to learn which signing processes it needs to simulate, the functionality makes the current list of queued (and finished) signing processes and their eids (preprocessing pointers) accessible to the simulator via $\mathcal{S}$.GetProcessInfo. This replaces the mechanism of $\mathcal{F}$-TS2, which informs the simulator every time Sign is called (at the expense of yielding control to the simulator, allowing it to interrupt the program flow, which $\mathcal{F}$-TSSync2 is trying to avoid).

Modeling general key generation processes. Like $\mathcal{F}$-TS3, we model key generation in $\mathcal{F}$-TSSync2 effectively as trusted (as it is expected to conclude without delay before the environment even calls KeyGen). One can extend $\mathcal{F}$-TSSync2 to model non-trusted key generation using the same approach as for signatures. KeyGen would keep track of which parties started key generation in the same round. Once enough parties sign up to generate a key, it will be made available via GetPk after $\mathsf{rnd}_{\text{key}}$ rounds. The simulator is expected to call SetPk before that, but if it fails, a default public key $\mathsf{pk} = 1$ is chosen to guarantee that the functionality will be able to create signatures.

The role of the execution ID eid. The execution ID eid can be chosen arbitrarily by the environment when calling Preprocess. The eid is only used to identify the resulting preprocessing value pp for the subsequent Sign call. In contrast to $\mathcal{F}$-TS2, the functionality $\mathcal{F}$-TSSync2 does not require parties to exchange eids, i.e., Sign does not require $(\mathsf{eid}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$ as input (only $(\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$). Instead, a joint signing session is logically identified by shared $(\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}, ssid, round_{\text{start}})$ values (see Line 53).

Realizing the functionality. In Appendix G, we present a protocol Π-TSSync2 that UC-realizes $\mathcal{F}$-TSSync2 using $\mathcal{F}$-TS3. The idea for this protocol is that whenever Tick() is called, it iterates over

[10] A signature is returned after $\mathsf{rnd}_{\text{sign}} - 1$ online rounds (instead of $\mathsf{rnd}_{\text{sign}}$ rounds as one might expect). This is because the online signing process effectively starts with round 2, as we consider round 1 to be preprocessing.

Functionality 9: $\mathcal{F}$-TSSync2$[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}]$

with accesss to $\mathcal{G}$-Clock.
The code below references $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ from Game 3 and ForceValidity from Functionality 4.

(⏲)Init()
SGN$[\cdot,\cdot] := \varnothing$ // Signers for unforgeability
Proc$[\cdot] := \varnothing$ // Ongoing signing processes
$\text{EID}_{\mathcal{P}}[\cdot] := \bot$ // eid for ongoing processes
SigRes$[\cdot,\cdot] := \bot$ // $(\mathcal{P}, proc) \mapsto \sigma$ result
$\text{PP}_{\mathcal{P}}[\cdot] := \bot$ for all $\mathcal{P} \in \mathbf{P}$ // Preprocessings
$\text{Used}_{\mathcal{P}} = \varnothing$ // Used eids
RndDone$[\cdot] := 0$ // $\mathcal{P} \mapsto$ latest round
VER$[\cdot,\cdot,\cdot] := \bot$ // Verify results
$\mathsf{Verify} := (\cdot \mapsto 0)$ // Default
$\mathsf{Preprocess} := [\cdot \mapsto (1,1)]$ // Default
$st_{\text{pp}} := ()$ // State for Preprocess
$\mathsf{pk} := \bot$
$(\mathsf{Verify}, \mathsf{Preprocess}, \mathsf{pk}') \leftarrow$ ⏲$\mathcal{S}$::GetAlg()
if $\mathsf{pk} = \bot$ **then**
  $\mathsf{pk} := \mathsf{pk}'$

$\mathcal{P}$.KeyGen() for $\mathcal{P} \in \mathbf{P}$
if $\mathsf{pk} = \bot$ **then** $\mathsf{pk} := 1$ // Default
return pk

$\mathcal{S}$.Corrupt($\mathcal{P}$)
assert $\mathcal{P} \in \mathbf{P}$
assert cor = adaptive $\vee$ [$\mathcal{P}$.KeyGen has not been called yet]
$\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
return () // No secret to reveal.

(⏲)$\mathcal{P}$.Tick() for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
RndDone$[\mathcal{P}] :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
(⏲)$\mathcal{G}$-Clock::$\mathcal{P}$.RoundOK()

$\mathcal{V}$.Verify(pk', m, σ) for any honest party $\mathcal{V} \notin \mathbf{C}$
$b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
if $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}', m)$ **then**
  $b := 0$ // Unforgeability
if $\text{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
  $\text{VER}[\mathsf{pk}', m, \sigma] := b$
return $\text{VER}[\mathsf{pk}', m, \sigma]$

$\mathsf{ValidateCorR}_{\mathbf{t}}(\mathcal{P}, proc, now)$
$(\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}}) := proc$
assert $\mathcal{P} \in \boldsymbol{T}$ and $\mathcal{P} \notin \mathbf{C}$ // Honest $\mathcal{P}$ participated
assert $\text{Proc}[proc] \supseteq \boldsymbol{T} \cap \mathbf{H}$ // Honest signers started
assert $now \geq round_{\text{start}} + \mathsf{rnd}_{\text{sign}} - 1$ // Finished
if $|\mathbf{C} \cap \boldsymbol{T}| = 0 \wedge |\boldsymbol{T}| \geq t_{\text{correct}}$ **then**
  return 1 // Correctness
if $|\mathbf{C}| < t_{\text{corrupt}} \wedge |\boldsymbol{T}| \geq t_{\text{robust}}$ **then**
  return 1 // Robustness
return 0

$\mathcal{P}$.Preprocess(eid) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
assert $\mathcal{P}$.KeyGen() has been called.
assert $\text{PP}_{\mathcal{P}}[\mathsf{eid}] = \bot$
$(\mathsf{pp}, st_{\text{pp}}) \leftarrow \mathsf{Preprocess}(\mathcal{P}, \mathsf{eid}, st_{\text{pp}})$
$\text{PP}_{\mathcal{P}}[\mathsf{eid}] := \mathsf{pp}$
return pp

$\mathcal{P}$.Sign($\mathsf{eid}, \boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
$\mathcal{G}$-Clock::$\mathcal{P}$.Register() // Ensure honest parties wait on $\mathcal{P}$ to finish round.
assert $\boldsymbol{T} \subseteq \mathbf{P} \wedge \mathcal{P} \in \boldsymbol{T}$
assert $\text{PP}_{\mathcal{P}}[\mathsf{eid}] = \mathsf{pp}_{\mathcal{P}} \neq \bot$
assert $\mathsf{eid} \notin \text{Used}_{\mathcal{P}}$
$\text{Used}_{\mathcal{P}} := \text{Used}_{\mathcal{P}} \cup \{\mathsf{eid}\}$
$ssid := \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$
$now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
assert $now > \text{RndDone}[\mathcal{P}]$ // $\mathcal{P}$.Tick() not yet called this round.
$round_{\text{start}} := now$ // Start this round.
$proc := (\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}})$
$\text{EID}_{\mathcal{P}}[proc] := \mathsf{eid}$
$\text{Proc}[proc] := \text{Proc}[proc] \cup \{\mathcal{P}\}$
$\text{SGN}[m, ssid] := \text{SGN}[m, ssid] \cup \{\mathcal{P}\}$
return $proc$

$\mathcal{P}$.GetSig($proc$) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
$(\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}}) := proc$
$now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
$\sigma := \text{SigRes}[\mathcal{P}, proc]$
if $\mathsf{ValidateCorR}_{\mathbf{t}}(\mathcal{P}, proc, now) = 1$ **then**
  $\sigma \leftarrow \mathsf{ForceValidity}(m, \sigma)$ // Correctness, Robustness
return σ

$\mathcal{S}$.GetProcessInfo()
return $(\text{Proc}, (\text{EID}_{\mathcal{P}})_{\mathcal{P} \in \mathbf{P}}, st_{\text{pp}})$

$\mathcal{S}$.SetPreprocState(st'_{pp})
$st_{\text{pp}} := st'_{\text{pp}}$

$\mathcal{S}$.SetSig($\mathcal{P}, proc, \sigma$)
$\text{SigRes}[\mathcal{P}, proc] := \sigma$

all currently running signing processes, reads messages from the previous round from the network $\mathcal{F}$-NetSync, calls the appropriate SIGN_k procedures of $\mathcal{F}$-TS3, and sends the resulting messages to the other signers via $\mathcal{F}$-NetSync. This protocol inherits the unforgeability, robustness, and correctness properties of $\mathcal{F}$-TS3, given that the network channel $\mathcal{F}$-NetSync is reliable and authenticated and hence guarantees the "protocol messages must have been properly delivered" preconditions for correctness ($\mathsf{ValidateC_t}$ in $\mathcal{F}$-TS3) and robustness ($\mathsf{ValidateR_t}$). We prove the following theorem in Appendix G.

Theorem 5. *The protocol Π-TSSync2$[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}]$ (uniform-)UC-realizes the ideal functionality $\mathcal{F}$-TSSync2$[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}]$ in the $\mathcal{F}$-NetSync and $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd} = (\mathsf{rnd}_{\text{T}} = 2, \mathsf{rnd}_{\text{msg}} = 2, \mathsf{rnd}_{\text{ctx}} = 2, \mathsf{rnd}_{\text{com}} = 2, \mathsf{rnd}_{\text{sign}}), \mathsf{ssid}, \mathsf{cor}]$ hybrid model in the presence of $\mathcal{G}$-Clock.*

8 A Simple UC Functionality for the Synchronous Setting

We present a functionality $\mathcal{F}$-TSSync1, a simplified version of $\mathcal{F}$-TSSync2 without preprocessing and only modeling the unforgeability level UF-1. Like $\mathcal{F}$-TS1, it is easy to modify $\mathcal{F}$-TSSync1 to model UF-0 security by changing the condition $|\text{SGN}[m] \cup \mathbf{C}| < t_{\text{UF}}$ in VERIFY to $\text{SGN}[m] \cap \mathbf{H} = \varnothing$

The functionality is listed in Functionality 10.

We prove the following theorem, establishing that $\mathcal{F}$-TSSync1 can be realized using $\mathcal{F}$-TS3.

Theorem 6. *The protocol Π-TSSync1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ (uniform-)UC-realizes $\mathcal{F}$-TSSync1$[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathbf{t}, \mathsf{cor}]$ in the $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl} = \text{UF-1}, \mathbf{rnd} = (\mathsf{rnd}_{\text{T}} = 1, \mathsf{rnd}_{\text{msg}} = 1, \mathsf{rnd}_{\text{ctx}} = 1, \mathsf{rnd}_{\text{com}} = 1, \mathsf{rnd}_{\text{sign}}), \mathsf{ssid} = (\cdot \mapsto 0), \mathsf{cor}]$ and $\mathcal{F}$-NetSync hybrid model in the presence of $\mathcal{G}$-Clock.*

The proof is a trivial simplification of the proof for Theorem 5. The protocol Π-TSSync1 is listed in Appendix H.

9 Adapting Our Results to Idealized Models

While the results in previous sections have been stated in the plain model, they can be adapted to schemes in the random oracle model (ROM) as well, by using a Global Random Oracle (GRO) in the UC framework [CDG+18, CJS14]. We present a variant of Theorem 2 in the presence of $\mathcal{G}$-RO (Functionality 11), which can be queried by any party and other ideal functionalities (regardless of the session they belong to). Importantly, we do not need any observability nor programmability of $\mathcal{G}$-RO, since our simulator $\mathcal{S}_{\text{TS3}}$ merely supplies the description of Σ_{TS}. This observation is analogous to the equivalence result for ordinary signature schemes in the ROM [CDG+18], which was also stated using $\mathcal{G}$-RO.

Theorem 7. *Let Σ_{TS} be a threshold signature scheme in the random oracle model, i.e., all of its algorithms have oracle access to a random oracle. Let Π-TS3$_{\text{RO}}$, $\mathcal{F}$-TS3$_{\text{RO}}$, $\mathcal{G}$-Setup$_{\text{RO}}$, and $\mathcal{F}$-Gen$_{\text{RO}}$ be identical to the corresponding counterparts in Theorem 2, except that whenever their procedure for $\mathcal{P}$ queries the random oracle with input x, it calls $y \leftarrow \mathcal{G}$-RO::$\mathcal{P}$.HASH(x) and responds with y. Then Π-TS3$_{\text{RO}}[\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}, \mathsf{cor}]$ (uniform-)UC-realizes $\mathcal{F}$-TS3$_{\text{RO}}[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ in the $\mathcal{F}$-Gen$_{\text{RO}}$ hybrid model in the presence of $\mathcal{G}$-RO, $\mathcal{G}$-Setup$_{\text{RO}}$ if and only if the threshold signature scheme $\Sigma_{\mathsf{TS}} = (\mathbf{rnd}, \mathsf{Setup}, \mathsf{KeyGen}, \{\mathsf{Sign}_k\}_{k=1}^{\mathsf{rnd}_{\text{sign}}}, \mathsf{Verify}, \mathsf{ssid})$ is $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-correct (Definition 1), $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-robust (Definition 3), and $(\mathsf{cor}, \mathsf{lvl}, \mathbf{P}, \mathbf{t})$-unforgeable (Definition 4).*

The proof of the above theorem is identical to that of Theorem 2, except that (1) the simulator $\mathcal{S}_{\text{TS3}}$ makes a query to $\mathcal{G}$-RO when Σ_{TS}::KeyGen calls the random oracle, (2) when showing the "if" direction, the reduction $\mathcal{B}$ to correctness, robustness, and unforgeability simulates $\mathcal{G}$-RO towards $\mathcal{A}$ and $\mathcal{Z}$ by making queries to its own random oracle in the game, and (3) when showing the "only if" direction, $\mathcal{Z}$ simulates the random oracle towards $\mathcal{B}$ against correctness, robustness, and unforgeability, by making queries to $\mathcal{G}$-RO. Theorems 3 and 4 follow immediately in the presence of $\mathcal{G}$-RO due to composability (Section 2.1).

Functionality 10: $\mathcal{F}$-TSSync1$[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathbf{t}, \mathsf{cor}]$

with access to $\mathcal{G}$-Clock. The code below references $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ from Game 3 and ForceValidity from Functionality 4.

(⏲)Init()
1: SGN$[\cdot,\cdot] := \varnothing$ // Signers for unforgeability
2: Proc$[\cdot] := \varnothing$ // Ongoing signing processes
3: SigRes$[\cdot,\cdot] := \bot$ // $(\mathcal{P}, proc) \mapsto \sigma$ result
4: VER$[\cdot,\cdot,\cdot] := \bot$ // Verify results
5: RndDone$[\cdot] := 0$ // $\mathcal{P} \mapsto$ latest round
6: $\mathsf{Verify} := (\cdot \mapsto 0)$ // Default
7: $\mathsf{pk} := \bot$
8: $(\mathsf{Verify}, \mathsf{pk}') \leftarrow \mathcal{S}$::GetAlg()
9: **if** $\mathsf{pk} = \bot$ **then**
10: $\quad \mathsf{pk} := \mathsf{pk}'$

$\mathcal{P}$.KeyGen() for $\mathcal{P} \in \mathbf{P}$
11: **if** $\mathsf{pk} = \bot$ **then** $\mathsf{pk} := 1$ // Default
12: **return** pk

$\mathcal{S}$.Corrupt($\mathcal{P}$)
13: **assert** $\mathcal{P} \in \mathbf{P}$
14: **assert** cor = adaptive $\vee$ [$\mathcal{P}$.KeyGen has not been called yet]
15: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
16: **return** () // No secret to reveal.

(⏲)$\mathcal{P}$.Tick() for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
17: RndDone$[\mathcal{P}] :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
18: (⏲)$\mathcal{G}$-Clock::$\mathcal{P}$.RoundOK()

$\mathcal{S}$.GetProcessInfo()
19: **return** Proc

$\mathcal{S}$.SetSig($\mathcal{P}, proc, \sigma$)
20: SigRes$[\mathcal{P}, proc] := \sigma$

$\mathcal{P}$.Sign($\boldsymbol{T}, m, ctx$) for $\mathcal{P} \in \mathbf{P}$
21: **assert** $\mathcal{P}$.KeyGen() has been called.
22: $\mathcal{G}$-Clock::$\mathcal{P}$.Register() // Ensure honest parties wait on $\mathcal{P}$ to finish round.
23: **assert** $\boldsymbol{T} \subseteq \mathbf{P} \wedge \mathcal{P} \in \boldsymbol{T}$
24: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
25: **assert** $now >$ RndDone$[\mathcal{P}]$ // $\mathcal{P}$.Tick() not yet called this round.
26: $round_{\text{start}} := now$ // Start this round.
27: $proc := (\boldsymbol{T}, m, ctx, round_{\text{start}})$
28: Proc$[proc] :=$ Proc$[proc] \cup \{\mathcal{P}\}$
29: SGN$[m] :=$ SGN$[m] \cup \{\mathcal{P}\}$
30: **return** $proc$

$\mathcal{P}$.GetSig($proc$) for $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$
31: $(\boldsymbol{T}, m, ctx, round_{\text{start}}) := proc$
32: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
33: $\sigma :=$ SigRes$[\mathcal{P}, proc]$
34: **if** $\mathsf{ValidateCorR}_{\mathbf{t}}(\mathcal{P}, proc, now) = 1$ **then**
35: $\quad \sigma \leftarrow \mathsf{ForceValidity}(m, \sigma)$ // Correctness, Robustness
36: **return** σ

$\mathsf{ValidateCorR}_{\mathbf{t}}(\mathcal{P}, proc, now)$
37: $(\boldsymbol{T}, m, ctx, round_{\text{start}}) := proc$
38: **assert** $\mathcal{P} \in \boldsymbol{T}$ and $\mathcal{P} \notin \mathbf{C}$ // Honest $\mathcal{P}$ participated
39: **assert** Proc$[proc] \supseteq \boldsymbol{T} \cap \mathbf{H}$ // Honest signers started
40: **assert** $now \geq round_{\text{start}} + \mathsf{rnd}_{\text{sign}}$ // Finished
41: **if** $|\mathbf{C} \cap \boldsymbol{T}| = 0 \wedge |\boldsymbol{T}| \geq t_{\text{correct}}$ **then**
42: $\quad$ **return** 1 // Correctness
43: **if** $|\mathbf{C}| < t_{\text{corrupt}} \wedge |\boldsymbol{T}| \geq t_{\text{robust}}$ **then**
44: $\quad$ **return** 1 // Robustness
45: **return** 0

$\mathcal{V}$.Verify(pk', m, σ) for any party $\mathcal{V}$
46: $b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
47: **if** $\mathsf{pk}' = \mathsf{pk} \wedge |\mathbf{C}| < t_{\text{corrupt}}$
48: $\wedge |\text{SGN}[m] \cup \mathbf{C}| < t_{\text{UF}}$ // UF-1 **then**
49: $\quad b := 0$
50: **if** VER$[\mathsf{pk}', m, \sigma] = \bot$ **then**
51: $\quad$ VER$[\mathsf{pk}', m, \sigma] := b$
52: **return** VER$[\mathsf{pk}', m, \sigma]$

Functionality 11: $\mathcal{G}$-RO[ℓ]

$\mathcal{G}$-RO is a global functionality parameterized by the output length ℓ.

INIT()
1: HT[$\cdot$] := $\perp$

$\mathcal{P}$.HASH(x) for all $\mathcal{P}$
2: **if** HT[x] = $\perp$ **then**
3: $y \leftarrow \{0,1\}^\ell$
4: HT[x] := y
5: **return** HT[x]

If the underlying threshold signature scheme Σ_{TS} is proven secure in the generic group model (GGM) [Sho97], our main theorems can be stated in the presence of a global generic group subroutine $\mathcal{G}$-GG (e.g., [BFKT24]). In this model, all machines across all sessions share a common group representation, and can only access and operate on the group elements encoded in random strings via $\mathcal{G}$-GG. Similar to the case of the ROM, our simulator $\mathcal{S}_{\mathsf{TS3}}$ does not need any observability nor programmability of $\mathcal{G}$-GG. The implication of our result is that schemes proven secure in the GGM can be securely composed with other protocols, as long as they are also proven secure in the presence of $\mathcal{G}$-GG.

Unlike the ROM and the GGM, our results do not have direct implications for Σ_{TS} proven secure in the algebraic group model (AGM) [FKL18]. Although there exists a composition framework for AGM [ABK+21], they establish a different framework than the original UC framework in which all machines, including simulators and an environment, are restricted to algebraic. While it may be possible to adapt our results to their framework or define a global subroutine $\mathcal{G}$-AG modeling an algebraic group within the standard UC framework, we leave it as future work.

Acknowledgments. We thank Pooya Farshim for discussing various ideal threshold signature functionalities and for many other insightful conversations. We thank the anonymous reviewers for their helpful comments. This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

ABK+21. M. Abdalla, M. Barbosa, J. Katz, J. Loss, and J. Xu. Algebraic adversaries in the universal composability framework. In *ASIACRYPT 2021, Part III*, vol. 13092 of *LNCS*, pp. 311–341. Springer, Cham, 2021.

ADMM15. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. On the malleability of Bitcoin transactions. In *FC 2015 Workshops*, vol. 8976 of *LNCS*, pp. 1–18. Springer, Berlin, Heidelberg, 2015.

ADN06. J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold RSA with adaptive and proactive security. In *EUROCRYPT 2006*, vol. 4004 of *LNCS*, pp. 593–611. Springer, Berlin, Heidelberg, 2006.

AF04a. M. Abe and S. Fehr. Adaptively secure feldman VSS and applications to universally-composable threshold cryptography. Cryptology ePrint Archive, Report 2004/119, 2004.

AF04b. M. Abe and S. Fehr. Adaptively secure feldman VSS and applications to universally-composable threshold cryptography. In *CRYPTO 2004*, vol. 3152 of *LNCS*, pp. 317–334. Springer, Berlin, Heidelberg, 2004.

AHS20. J.-P. Aumasson, A. Hamelink, and O. Shlomovits. A survey of ECDSA threshold signing. Cryptology ePrint Archive, Report 2020/1390, 2020.

AS20. J.-P. Aumasson and O. Shlomovits. Attacking threshold wallets. Cryptology ePrint Archive, Report 2020/1052, 2020.

BCdP+25. G. Borin, S. Celi, R. del Pino, T. Espitau, G. Niot, and T. Prest. Threshold signatures reloaded: ML-DSA and enhanced raccoon with identifiable aborts. Cryptology ePrint Archive, Paper 2025/1166, 2025.

BCH+20. C. Badertscher, R. Canetti, J. Hesse, B. Tackmann, and V. Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In *TCC 2020, Part III*, vol. 12552 of *LNCS*, pp. 1–30. Springer, Cham, 2020.

BCK+22. M. Bellare, E. C. Crites, C. Komlo, M. Maller, S. Tessaro, and C. Zhu. Better than advertised security for non-interactive threshold signatures. In *CRYPTO 2022, Part IV*, vol. 13510 of *LNCS*, pp. 517–550. Springer, Cham, 2022.

BdCE+25. A. Bienstock, L. de Castro, D. Escudero, A. Polychroniadou, and A. Takahashi. Efficient, scalable threshold ML-DSA signatures: An MPC approach. Cryptology ePrint Archive, Paper 2025/1163, 2025.

BDF+18. C. Brzuska, A. Delignat-Lavaud, C. Fournet, K. Kohbrok, and M. Kohlweiss. State separation for code-based game-playing proofs. In *ASIACRYPT 2018, Part III*, vol. 11274 of *LNCS*, pp. 222–249. Springer, Cham, 2018.

BDLR25a. R. Bacho, S. Das, J. Loss, and L. Ren. Adaptively secure three-round threshold Schnorr signatures from DDH. In *CRYPTO 2025, Part VI*, vol. 16005 of *LNCS*, pp. 390–422. Springer, Cham, 2025.

BDLR25b. R. Bacho, S. Das, J. Loss, and L. Ren. Glacius: Threshold schnorr signatures from DDH with full adaptive security. In *EUROCRYPT 2025, Part II*, vol. 15602 of *LNCS*, pp. 304–334. Springer, Cham, 2025.

BFKT24. J. Bobolz, P. Farshim, M. Kohlweiss, and A. Takahashi. The brave new world of global generic groups and UC-secure zero-overhead SNARKs. In *TCC 2024, Part I*, vol. 15364 of *LNCS*, pp. 90–124. Springer, Cham, 2024.

BKL+25. C. Boschini, D. Kaviani, R. Lai, G. Malavolta, A. Takahashi, and M. Tibouchi. Ringtail: Practical Two-Round Threshold Signatures from Learning with Errors . In *2025 IEEE Symposium on Security and Privacy*, pp. 149–164. IEEE Computer Society, 2025.

BL13. D. J. Bernstein and T. Lange. Non-uniform cracks in the concrete: The power of free precomputation. In *ASIACRYPT 2013, Part II*, vol. 8270 of *LNCS*, pp. 321–340. Springer, Berlin, Heidelberg, 2013.

BLL+22. F. Benhamouda, T. Lepoint, J. Loss, M. Orrù, and M. Raykova. On the (in)security of ROS. *Journal of Cryptology*, 35(4):25, 2022.

BLS04. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

BLT+24. R. Bacho, J. Loss, S. Tessaro, B. Wagner, and C. Zhu. Twinkle: Threshold signatures from DDH with full adaptive security. In *EUROCRYPT 2024, Part I*, vol. 14651 of *LNCS*, pp. 429–459. Springer, Cham, 2024.

Bol03. A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *PKC 2003*, vol. 2567 of *LNCS*, pp. 31–46. Springer, Berlin, Heidelberg, 2003.

BP25. L. Brandão and R. Peralta. Nist first call for multi-party threshold schemes. `https://nvlpubs.nist.gov/nistpubs/ir/2025/NIST.IR.8214C.2pd.pdf`, 2025.

BR06. M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT 2006*, vol. 4004 of *LNCS*, pp. 409–426. Springer, Berlin, Heidelberg, 2006.

BTZ22. M. Bellare, S. Tessaro, and C. Zhu. Stronger security for non-interactive threshold signatures: BLS and FROST. Cryptology ePrint Archive, Report 2022/833, 2022.

Can00. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000.

Can01. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pp. 136–145. IEEE Computer Society Press, 2001.

Can04. R. Canetti. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, p. 219. IEEE Computer Society, 2004.

Can20. R. Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.

CDG+18. J. Camenisch, M. Drijvers, T. Gagliardoni, A. Lehmann, and G. Neven. The wonderful world of global random oracles. In *EUROCRYPT 2018, Part I*, vol. 10820 of *LNCS*, pp. 280–312. Springer, Cham, 2018.

CDL+25. R. Cohen, J. Doerner, E. Lee, A. Lysyanskaya, and L. Roy. An unstoppable ideal functionality for signatures and a modular analysis of the dolev-strong broadcast. In *TCC 2025, Part IV*, vol. 16271 of *LNCS*, pp. 675–708. Springer, Cham, 2025.

CdPE+26. S. Celi, R. del Pino, T. Espitau, G. Niot, and T. Prest. Efficient threshold ML-DSA. In *USENIX Security Symposium (USENIX Security 2026)*, 2026.

CDPW07. R. Canetti, Y. Dodis, R. Pass, and S. Walfish. Universally composable security with global setup. In *TCC 2007*, vol. 4392 of *LNCS*, pp. 61–85. Springer, Berlin, Heidelberg, 2007.
CFOS25. G. Cho, G. Fuchsbauer, A. O'Neill, and M. Sefranek. Schnorr signatures are tightly secure in the ROM under a non-interactive assumption. In *CRYPTO 2025, Part VI*, vol. 16005 of *LNCS*, pp. 223–255. Springer, Cham, 2025.
CGG+20. R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *ACM CCS 2020*, pp. 1769–1787. ACM Press, 2020.
CGG+21. R. Canetti, R. Gennaro, S. Goldfeder, N. Makriyannis, and U. Peled. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. Cryptology ePrint Archive, Report 2021/060, 2021.
CGRS23. H. Chu, P. Gerhart, T. Ruffing, and D. Schröder. Practical Schnorr threshold signatures without the algebraic group model. In *CRYPTO 2023, Part I*, vol. 14081 of *LNCS*, pp. 743–773. Springer, Cham, 2023.
Che25. Y. Chen. Round-efficient adaptively secure threshold signatures with rewinding. *IACR Commun. Cryptol.*, 2(2):16, 2025.
CJS14. R. Canetti, A. Jain, and A. Scafuro. Practical UC security with a global random oracle. In *ACM CCS 2014*, pp. 597–608. ACM Press, 2014.
CKGW22. D. Connolly, C. Komlo, I. Goldberg, and C. Wood. Two-round threshold schnorr signatures with frost. https://datatracker.ietf.org/doc/draft-irtf-cfrg-frost/, 2022.
CKK+25. E. C. Crites, J. Katz, C. Komlo, S. Tessaro, and C. Zhu. On the adaptive security of FROST. In *CRYPTO 2025, Part VI*, vol. 16005 of *LNCS*, pp. 480–511. Springer, Cham, 2025.
CKM21. E. Crites, C. Komlo, and M. Maller. How to prove Schnorr assuming Schnorr: Security of multi- and threshold signatures. Cryptology ePrint Archive, Report 2021/1375, 2021.
CKM23. E. C. Crites, C. Komlo, and M. Maller. Fully adaptive Schnorr threshold signatures. In *CRYPTO 2023, Part I*, vol. 14081 of *LNCS*, pp. 678–709. Springer, Cham, 2023.
CMP20. R. Canetti, N. Makriyannis, and U. Peled. UC non-interactive, proactive, threshold ECDSA. Cryptology ePrint Archive, Report 2020/492, 2020.
CPZ24. C. Cremers, A. Peltonen, and M. Zhao. An extended hierarchy of security notions for threshold signature schemes and automated analysis of protocols that use them. Cryptology ePrint Archive, Report 2024/1920, 2024.
CS25. E. C. Crites and A. Stewart. A plausible attack on the adaptive security of threshold Schnorr signatures. In *CRYPTO 2025, Part VI*, vol. 16005 of *LNCS*, pp. 457–479. Springer, Cham, 2025.
Des88. Y. Desmedt. Society and group oriented cryptography: A new concept. In *CRYPTO'87*, vol. 293 of *LNCS*, pp. 120–127. Springer, Berlin, Heidelberg, 1988.
DF90. Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *CRYPTO'89*, vol. 435 of *LNCS*, pp. 307–315. Springer, New York, 1990.
DKLs18. J. Doerner, Y. Kondi, E. Lee, and a. shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy*, pp. 980–997. IEEE Computer Society Press, 2018.
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*, pp. 1051–1066. IEEE Computer Society Press, 2019.
DKLs24. J. Doerner, Y. Kondi, E. Lee, and a. shelat. Threshold ECDSA in three rounds. In *2024 IEEE Symposium on Security and Privacy*, pp. 3053–3071. IEEE Computer Society Press, 2024.
DKM+24. R. Del Pino, S. Katsumata, M. Maller, F. Mouhartem, T. Prest, and M.-J. O. Saarinen. Threshold raccoon: Practical threshold signatures from standard lattice assumptions. In *EUROCRYPT 2024, Part II*, vol. 14652 of *LNCS*, pp. 219–248. Springer, Cham, 2024.
dPKPR24. R. del Pino, S. Katsumata, T. Prest, and M. Rossi. Raccoon: A masking-friendly signature proven in the probing model. In *CRYPTO 2024, Part I*, vol. 14920 of *LNCS*, pp. 409–444. Springer, Cham, 2024.
DR24. S. Das and L. Ren. Adaptively secure BLS threshold signatures from DDH and co-CDH. In *CRYPTO 2024, Part VII*, vol. 14926 of *LNCS*, pp. 251–284. Springer, Cham, 2024.
DW14. C. Decker and R. Wattenhofer. Bitcoin transaction malleability and MtGox. In *ESORICS 2014, Part II*, vol. 8713 of *LNCS*, pp. 313–326. Springer, Cham, 2014.
EKT24. T. Espitau, S. Katsumata, and K. Takemure. Two-round threshold signature from algebraic one-more learning with errors. In *CRYPTO 2024, Part VII*, vol. 14926 of *LNCS*, pp. 387–424. Springer, Cham, 2024.
ENP24. T. Espitau, G. Niot, and T. Prest. Flood and submerse: Distributed key generation and robust threshold signature from lattices. In *CRYPTO 2024, Part VII*, vol. 14926 of *LNCS*, pp. 425–458. Springer, Cham, 2024.
FKL18. G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In *CRYPTO 2018, Part II*, vol. 10992 of *LNCS*, pp. 33–62. Springer, Cham, 2018.

GMR88. S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17(2):281–308, 1988.

Gro21. J. Groth. Non-interactive distributed key generation and key resharing. Cryptology ePrint Archive, Report 2021/339, 2021.

GS22. J. Groth and V. Shoup. Design and analysis of a distributed ECDSA signing service. Cryptology ePrint Archive, Report 2022/506, 2022.

KG20. C. Komlo and I. Goldberg. FROST: Flexible round-optimized Schnorr threshold signatures. In *SAC 2020*, vol. 12804 of *LNCS*, pp. 34–65. Springer, Cham, 2020.

KKL21. D. Karakostas, A. Kiayias, and M. Larangeira. Conclave: A collective stake pool protocol. In *ESORICS 2021, Part I*, vol. 12972 of *LNCS*, pp. 370–389. Springer, Cham, 2021.

KMTZ13. J. Katz, U. Maurer, B. Tackmann, and V. Zikas. Universally composable synchronous computation. In *TCC 2013*, vol. 7785 of *LNCS*, pp. 477–498. Springer, Berlin, Heidelberg, 2013.

KRT24. S. Katsumata, M. Reichle, and K. Takemure. Adaptively secure 5 round threshold signatures from MLWE/MSIS and DL with rewinding. In *CRYPTO 2024, Part VII*, vol. 14926 of *LNCS*, pp. 459–491. Springer, Cham, 2024.

Lin17. Y. Lindell. Fast secure two-party ECDSA signing. In *CRYPTO 2017, Part II*, vol. 10402 of *LNCS*, pp. 613–644. Springer, Cham, 2017.

Lin24. Y. Lindell. Simple three-round multiparty Schnorr signing with full simulatability. *CiC*, 1(1):25, 2024.

LNÖ25. A. Lehmann, P. Nazarian, and C. Özbay. Stronger security for threshold blind signatures. In *EUROCRYPT 2025, Part II*, vol. 15602 of *LNCS*, pp. 335–364. Springer, Cham, 2025.

LW16. J. Lau and P. Wuille. Dealing with signature encoding malleability. bitcoin improvement proposal 146. `https://github.com/bitcoin/bips/blob/master/bip-0146.mediawiki`, 2016.

Mak22. N. Makriyannis. On the classic protocol for MPC Schnorr signatures. Cryptology ePrint Archive, Report 2022/1332, 2022.

NT24. S. Navot and S. Tessaro. One-more unforgeability for multi - and threshold signatures. In *ASIACRYPT 2024, Part I*, vol. 15484 of *LNCS*, pp. 429–462. Springer, Singapore, 2024.

PKN+25. R. D. Pino, S. Katsumata, G. Niot, M. Reichle, and K. Takemure. Unmasking TRaccoon: A lattice-based threshold signature with an efficient identifiable abort protocol. In *CRYPTO 2025, Part VI*, vol. 16005 of *LNCS*, pp. 423–456. Springer, Cham, 2025.

RRJ+22. T. Ruffing, V. Ronge, E. Jin, J. Schneider-Bensch, and D. Schröder. ROAST: Robust asynchronous Schnorr threshold signatures. In *ACM CCS 2022*, pp. 2551–2564. ACM Press, 2022.

Sho97. V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97*, vol. 1233 of *LNCS*, pp. 256–266. Springer, Berlin, Heidelberg, 1997.

Sho00. V. Shoup. Practical threshold signatures. In *EUROCRYPT 2000*, vol. 1807 of *LNCS*, pp. 207–220. Springer, Berlin, Heidelberg, 2000.

Sho25. V. Shoup. The many faces of Schnorr: a toolkit for the modular design of threshold Schnorr signatures. *CiC*, 2(1):18, 2025.

WNR20. P. Wuille, J. Nick, and T. Ruffing. Schnorr signatures for secp256k1. bitcoin improvement proposal 340. `https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki`, 2020.

Wui14. P. Wuille. Dealing with malleability. bitcoin improvement proposal 62. `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`, 2014.

ZT25. C. Zhu and S. Tessaro. The algebraic one-more MISIS problem and applications to threshold signatures. In *CRYPTO 2025, Part I*, vol. 16000 of *LNCS*, pp. 548–581. Springer, Cham, 2025.

A Related work

Table 4 gives an overview of both our definitions and related work. We discuss details and other related work below.

A.1 Comparison with other Game-based Hierarchies for Threshold Signatures

We first provide an overview of other hierarchies of unforgeability notions for threshold signature schemes.

The standard notion of unforgeability for threshold signatures (coined TS-UF-0 security by Bellare et al. [BTZ22, BCK+22]) is the natural analogue of EUF-CMA security for single-party signature schemes [GMR88]. A forgery is considered trivial if even one honest party was asked to produce a partial signature on the forgery message. Most threshold signature schemes in the literature are proven under this notion, cf. Table 1.

Shoup. Shoup [Sho00] introduces a stronger notion of security for non-interactive threshold signatures, which considers a forgery non-trivial if a partial signature on the forgery message was requested from at most $t_{\text{correct}} - t_{\text{corrupt}}$ honest servers, where t_{correct} is the threshold, and t_{corrupt} is the minimum number of corrupt parties that voids the unforgeability. This is TS-UF-1 security in the formalism of Bellare et al.

Groth. Groth [Gro21] gives a general definition for non-interactive threshold signature schemes with a (non-interactive) distributed key generation (DKG) protocol, which implies TS-UF-1 security.

Bellare et al. Bellare et al. [BTZ22, BCK+22] present a hierarchy, TS-UF-0 to 4 and TS-SUF-2 to 4, of unforgeability notions for non-interactive and partially non-interactive threshold signature schemes, where the message and signing set are determined in the (single) online signing round. Multi-round protocols with more than one online round or more than one offline, preprocessing round are out of scope. Our game-based security hierarchy implies the hierarchy of Bellare et al., and thus their findings, including strong unforgeability, translate to our setting. However, we show that threshold BLS [Bol03] actually achieves a higher security notion (SUF-2 rather than UF-1), reinstating the strong unforgeability expected of a unique signature scheme.

Cremers et al. Cremers et al. [CPZ24] build on the hierarchy of Bellare et al., capturing multi-round protocols, including distributed key generation. Importantly, their hierarchy includes security in the adaptive corruption setting. Our hierarchy implies the hierarchy of Cremers et al. assuming idealized key generation. As in Bellare et al., strong unforgeability for schemes without preprocessing is not considered. This is a gap we fill in this work. Moreover, we do not include their strong unforgeability levels SUF-0 and SUF-1 due to their malleability, discussed in Appendix C. We note that some protocol designers are opting to prove much lower levels of security than their schemes likely achieve due to the subtleties involved in defining security with leader requests (inherited from [BTZ22, BCK+22]) for multi-round protocols [KRT24]. We remove this barrier, defining leader requests as a special case of our full session ID classification in Section 3.4.

Navot-Tessaro. Navot and Tessaro [NT24] explore one-more unforgeability as a way of modeling strong unforgeability for threshold signatures and multi-signatures. The strong unforgeability notion is a significantly weakened version of the definition in Bellare et al. (implied by TS-SUF-2), however validated by the existence of a scheme that does not meet it [RRJ+22].

Lehmann et al. Lehmann et al. [LNÖ25] define a hierarchy of one-more unforgeability notions for threshold blind signatures, capturing non-interactive and interactive protocols in the static corruption setting. Their hierarchy roughly follows the UF-x hierarchy of Bellare et al., but deviates where definitions are not relevant in the blind setting. For example, the distinction between UF-3 and UF-4 and the trivial lifting of UF-0 to UF-1 security by the standard guessing argument do not apply.

A.2 Comparison with other UC Functionalities for Threshold Signatures

We now provide an overview of different UC functionalities for threshold signatures in the literature.

$\mathcal{F}$-`tSig`–style functionalities. The approach of modeling threshold signature functionalities in the $\mathcal{F}$-`tSig` definitional style is not new to our paper. Earlier work [CGG+21, CGG+20] describes an ideal functionality for threshold signatures. Similar to our functionalities, their functionality models

	Games $\mathcal{F}$-TS3	$\mathcal{F}$-TS2	$\mathcal{F}$-TS1	$\mathcal{F}$-TSSync2	$\mathcal{F}$-TSSync1	[CGG+21]	[CDL+25]	[BTZ22]	[CPZ24]
Type	Game/UC	UC	UC	UC	UC	UC	UC	Game	Game
$\mathsf{rnd}_{\mathsf{sign}}$	Param.	∞	∞	Param.	Param.	∞	Param.	2	Param.
Preprocessing	Param.	1 round	No	1 round	No	No	No	1 round	$\leq$ 1 round
Corruption	Param.	Param.	Param.	Param.	Param.	Mobile	Adaptive	Static	Param.
KeyGen	Trusted	Agnostic	Agnostic	Trusted	Trusted	Agnostic	Sim	Trusted	Prot
Correctness	Comp	No	No	Comp	Comp	No	Comp	Comp	Comp
Robustness	Param.	No	No	Param.	Param.	IA	Yes	No	No
Unforgeability	(S)UF-X	(S)UF-X	UF-1	(S)UF-X	UF-1	UF-2	Sim	((S)UF-X)	((S)UF-X)
ssid	Param.	Param.	Irrelevant	Param.	Irrelevant	ctx	ctx	$\boldsymbol{T}, ctx, \mathsf{pp}_{\boldsymbol{T}}$	$\boldsymbol{T}, ctx, \mathsf{pp}_{\boldsymbol{T}}$
Network	Insecure	Agnostic	Agnostic	Agnostic	Agnostic	Agnostic	Agnostic	Insecure	Param.
Notes				Clock	Clock	Key Rfrsh	Clock		

Table 4: Comparison of threshold signature definitions. Entries denote the least desirable value a scheme can have and still be considered secure under the given definition. "Param." means that the definition can be parameterized to restrict schemes to any desired value. The value implying the strongest possible property is generally marked in green, the weakest possible property in red. "Agnostic" means that the definition does enforce anything in this category. "Comp" means computational correctness (i.e., the scheme is correct except with negligible probability). "Prot" means that the key generation is modeled as a multi-round protocol that is part of the scheme. "Clock" means that the functionality uses a clock to model synchronous communication. "Key Rfrsh" and "Mobile" corruption means that the functionality supports key refreshes and the adversary can uncorrupt parties.

a threshold signature functionality with explicit unforgeability guarantees rather than an MPC-style functionality that jointly computes digital signatures. The functionality counts the number of parties that want to sign a certain message m in a given user-defined session "sgid" (which we call "*ssid*") and asks the simulator to provide a signature. The verification procedure ensures that no signature on m is valid unless enough parties have requested a signature on m on some shared session *ssid* (corresponding to security level UF-2 with $\mathsf{ssid}(\cdot) = ctx$ in our notation, see Section 3). Their functionality is most closely related to our $\mathcal{F}$-TS2 functionality, which supports preprocessing and is more general but can be paramterized to achieve very similar properties. The [CGG+21] functionality supports key refreshes (for proactive security) and allows the adversary to uncorrupt parties, which we do not consider in our paper. Several similar versions of this functionality with small variations can be found in [Mak22, CMP20, CGG+20]. Additionally, there are several other works that use a similar style of functionality, such as [AF04a, ADN06, KKL21].

Unstoppable $\mathcal{F}$-tSig. [CDL+25] presents a very thorough definition of an ideal threshold signature functionality with strong simulation-based unforgeability and with robustness guarantees. Keeping to their theme of an unstoppable functionality, they use a synchronous communication model [KMTZ13] to ensure that the adversary cannot prevent the functionality/protocol from producing a signature. We use the same mechanism in our $\mathcal{F}$-TSSync1 and $\mathcal{F}$-TSSync2 functionalities. Interestingly, while their definition mostly follows the $\mathcal{F}$-tSig paradigm (they explicitly model unforgability guarantees), their functionality generates a target signature σ^* and the simulator must simulate the signing protocol to hit this signature, which is more akin to the strong $\mathcal{F}$-Sign style definitions, as we discussed above. Because of this, combined with their strong robustness requirements, their functionality is difficult to realize. Importantly, they note [CDL+25] several weaknesses in existing (non-threshold) signature functionalities and describe solutions for them. We follow ideas clearly inspired by their work. See Functionality 12 for examples.

$\mathcal{F}$-Sign–style functionalities. $\mathcal{F}$-Sign–style functionalities do not directly model an ideal threshold signature functionality, but rather they model simulation-based security of an MPC protocol to jointly compute a digital signature from some underlying scheme in a threshold manner. This approach has been taken, for example, for ECDSA [Lin17, DKLs18, DKLs19, GS22, DKLs24], Schnorr [Lin24], and ML-DSA [BdCE+25] signatures. We discuss the advantages and disadvantages of this definitional approach in the introduction. Generally, a scheme secure in the $\mathcal{F}$-Sign model (assuming unforgeable underlying signatures) will also be unforgeable in our (less demanding) game-based and UC definitions, though details depend on the concrete protocol (e.g., some of our definitions enforce a fixed number of communication rounds, which $\mathcal{F}$-Sign functionalities usually do not).

B Warm-up: Digital Signature Functionality

We illustrate our syntax for UC programs by describing a digital signature functionality $\mathcal{F}$-Sig in Fig. 12.

It is similar to Canetti's signature functionality, but we have incorporated a simplified version of the random mode from [CDL+25]. It defines a collection of procedures that can be called by a single signing party $\mathcal{P}^*$: KeyGen and Sign. It also defines a procedure Verify that can be called by any party $\mathcal{P}$.

Functionality 12: $\mathcal{F}$-Sig$[\mathcal{P}^*]$

For a single signer party $\mathcal{P}^*$.

(⏲)Init()

$\mathrm{SGN}[\cdot] := 0$
$\mathrm{VER}[\cdot,\cdot,\cdot] := \bot$
$\mathsf{Sign} := (\cdot \mapsto 0)$ // Default signature 0.
$\mathsf{Verify} := (\cdot \mapsto 0)$ // Reject all.
$\mathsf{pk} := \bot$
$(\mathsf{Sign}, \mathsf{Verify}, \mathsf{pk}') \leftarrow$ ⏲$\mathcal{S}$::GetAlg()
if $\mathsf{pk} = \bot$ **then**
  $\mathsf{pk} := \mathsf{pk}'$

$\mathcal{P}^*$.KeyGen()

if $\mathsf{pk} = \bot$ **then** $\mathsf{pk} := 1$ // Default
return pk

ForceValidity(m, σ) (internal)

if $\mathrm{VER}[\mathsf{pk}, m, \sigma] = 0$ **then**
  $\sigma \leftarrow_{\text{first}} \{\sigma' \in \{0,1\}^* \mid \mathrm{VER}[\mathsf{pk}, m, \sigma'] \neq 0\}$
$\mathrm{VER}[\mathsf{pk}, m, \sigma] := 1$
return σ

$\mathcal{P}^*$.Sign(m)

assert $\mathcal{P}^*$.KeyGen() has been called.
$\mathrm{SGN}[m] \leftarrow \mathrm{SGN}[m] + 1$
$\sigma \leftarrow \mathsf{Sign}(m)$
$\sigma \leftarrow$ ForceValidity(m, σ)
return σ

$\mathcal{P}$.Verify$(\mathsf{pk}', m, \sigma)$ for all $\mathcal{P}$

$b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
if $\mathsf{pk}' = \mathsf{pk} \wedge \mathrm{SGN}[m] = 0$ **then**
  $b \leftarrow 0$ // Unforgeability
if $\mathsf{pk}' = \mathsf{pk} \wedge \mathrm{SGN}[m] = |\{\sigma' \mid \mathrm{VER}[\mathsf{pk}, m, \sigma'] = 1\}|$ **then**
  $b \leftarrow 0$ // Strong unforgeability
if $\mathrm{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
  $\mathrm{VER}[\mathsf{pk}', m, \sigma] \leftarrow b$
return $\mathrm{VER}[\mathsf{pk}', m, \sigma]$

$\mathcal{S}$.Corrupt$(\mathcal{P})$

assert $\mathcal{P} = \mathcal{P}^*$
assert $\mathsf{cor} = \text{adaptive} \vee [\mathcal{P}^*$.KeyGen has not been called yet$]$
$\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}^*\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}^*\}$
return ()

Upon initialization, $\mathcal{F}$-Sig receives from the simulator $\mathcal{S}$ a signing algorithm Sign, a verification algorithm Verify, and a public key pk. Here, we mark the procedure $\mathcal{S}$::GetAlg() with (⏲) to indicate that the simulator may suspend the execution of $\mathcal{F}$-Sig to the environment before returning to $\mathcal{F}$-Sig. Since $\mathcal{S}$ can potentially give control to $\mathcal{Z}$ before returning, the procedures of $\mathcal{F}$-Sig are accessible and can be called before (Sign, Verify, pk) are set by $\mathcal{S}$. This is why $\mathcal{F}$-Sig sets default values for (Sign, Verify, pk), ensuring that $\mathcal{F}$-Sig is always in a well-defined state.

The internal procedure ForceValidity is used to ensure that signatures produced by $\mathcal{F}$-Sig are always accepted by the verification procedure. If the signature σ produced by $\mathsf{Sign}(m)$ has previously been marked as invalid in VER, then ForceValidity replaces it with some signature that is accepted by Verify, similar to the random mode in [CDL+25]. Note that ForceValidity here is simpler than in Functionality 4 due to the single party setting.

Moreover, Verify ensures that signatures are unforgeable by rejecting signatures for messages that have not been signed. It ensures strong unforgeability by counting the expected number of (different) signatures on m, and rejecting additional signatures.

C A Remark on SUF-0 and SUF-1

Note the absence of SUF-0 and SUF-1 levels in Table 3. As noted in Section 3.4, this is because we need *ssid* to meaningfully group signing processes together and limit the number of expected signatures $\mathrm{NumSig}_{\mathsf{lvl}, t_{\mathrm{UF}}}$ to the number of different *ssid*.

[CPZ24] does define these levels, but their definition is quite weak and does not capture the intuitive goal of strong unforgeability. They call $(\boldsymbol{T}, m, ctx)$ the *leader request lr* and strong security is defined, roughly, by saying that for every *lr*, there shall exist a *unique* signature σ accepted by a *strong verification* algorithm SVrfy.

As an example to showcase the weakness of this notion, consider a (threshold) signature scheme where $\mathsf{Sign}(\boldsymbol{T}, m, ctx)$ runs normal BLS signing and outputs a signature of the format $(\mathsf{BLS.Sign}(m), \boldsymbol{T}, ctx)$, i.e., our scheme appends $\boldsymbol{T}, ctx$ to the unique BLS signature on m. The verification algorithm Verify for our contrived scheme simply ignores the $\boldsymbol{T}, ctx$ part of the signature. The strong verification algorithm $\mathsf{SVrfy}(lr = (\boldsymbol{T}, m, ctx), \sigma)$ additionally checks that $\sigma = (\cdot, \boldsymbol{T}, ctx)$, i.e., that the $\boldsymbol{T}, ctx$ in the leader request corresponds to the ones indicated in the signature.

This scheme would be considered SUF-0 and SUF-1 unforgeable (assuming an underlying interactive protocol for BLS with UF-1 security): For every leader request $lr = (\boldsymbol{T}, m, ctx)$, there indeed exists a unique signature $\sigma = (\mathsf{BLS.Sign}(m), \boldsymbol{T}, ctx)$ that is accepted by the strong verification algorithm SVrfy (using that BLS has unique signatures). The other part of the unforgeability requirements is covered by UF-1 unforgeability of the underlying BLS signing protocol.

However, clearly, our contrived signature scheme is not strongly unforgeable in any meaningful way: signatures are highly malleable. After observing the signature σ of a single signing process, it is trivial to modify σ to another signature $\sigma' \neq \sigma$ on the same message still accepted by Verify, simply by arbitrarily changing the $(\boldsymbol{T}, ctx)$ part of σ. In other words, after observing a single signature, one can trivially produce an infinite number of valid signatures.

D Proof of Theorem 2 (Π-TS3 UC-realizes $\mathcal{F}$-TS3)

In this section, we prove the following theorem:

Theorem 2 (restated). The protocol Π-TS3$[\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}, \mathsf{cor}]$ (uniform-)UC-realizes $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ in the $\mathcal{F}$-Gen$[\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}::\mathsf{Setup}, \Sigma_{\mathsf{TS}}::\mathsf{KeyGen}]$ hybrid model in the presence of $\mathcal{G}$-Setup $[\Sigma_{\mathsf{TS}}::\mathsf{Setup}]$ if and only if the threshold signature scheme $\Sigma_{\mathsf{TS}} = (\mathbf{rnd}, \mathsf{Setup}, \mathsf{KeyGen}, \{\mathsf{Sign}_k\}_{k=1}^{\mathsf{rnd}_{\mathrm{sign}}}, \mathsf{Verify}, \mathsf{ssid})$ is $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-correct (Definition 1), $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$-robust (Definition 3), and $(\mathsf{cor}, \mathsf{lvl}, \mathbf{P}, \mathbf{t})$-unforgeable (Definition 4).

Proof. **Proof for "if" direction** We define the UC simulator $\mathcal{S}_{\text{TS3}}$ in Simulator 1. We then assume that the simulator $\mathcal{S}$ of the ideal world experiment interacting with $\mathcal{F}$-TS3 is the composition of $\mathcal{S}_{\text{TS3}}$ and $\mathcal{A}$.

Simulator 1: $\mathcal{S}_{\text{TS3}}$

Note that $\mathcal{A}$ calls $\mathcal{S}_{\text{TS3}}$ internally to simulate corruption.

GetAlg()

1: // Generate keys like Π-TS3 would (via $\mathcal{F}$-Gen)
2: $\mathsf{par} \leftarrow \mathcal{G}\text{-Setup}[\mathsf{Setup}]::\mathcal{P}.\textsc{Setup}()$
3: $(\mathsf{pk}, \{\mathsf{sk}_{\mathcal{P}}\}_{\mathcal{P}\in\mathbf{P}}) \leftarrow \mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})$
4: // Set initial state like Π-TS3 would
5: $\mathsf{st}_{\mathcal{P}} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_{\mathcal{P}})$
6: // Return honest $\mathsf{Verify}, \mathsf{Sign}_k$ algorithms and keys to $\mathcal{F}$-TS3
7: **return** $(\Sigma_{\mathsf{TS}}::\mathsf{Verify}, (\Sigma_{\mathsf{TS}}::\mathsf{Sign}_k)_{k=1}^{\mathsf{rnd}_{\mathrm{sign}}}, \Sigma_{\mathsf{TS}}::\mathsf{Combine}, \mathsf{pk}, (\mathsf{st}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}})_{\mathcal{P}\in\boldsymbol{T}})$ to $\mathcal{F}$-TS3

Π-TS3$_{\mathcal{P}}$::$\mathcal{A}$.Corrupt()

8: $(\mathsf{sk}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P}}, \mathrm{ST}_{\mathcal{P}}) \leftarrow \mathcal{F}\text{-TS3}::\mathcal{S}.\textsc{Corrupt}(\mathcal{P})$
9: **return** $(\mathsf{sk}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P}}, \mathrm{ST}_{\mathcal{P}})$ to $\mathcal{A}$

We prove the statement via the following hybrids.

$\mathsf{Hyb}_0 = \mathsf{EXEC}_{\Pi\text{-TS3},\mathcal{G}\text{-Setup},\mathcal{A},\mathcal{Z}}$ This hybrid corresponds to a real execution of Π-TS3 in the $\mathcal{F}$-Gen-hybrid model (in the presence of global subroutine $\mathcal{G}$-Setup).

$\mathsf{Hyb}_1 = \mathsf{EXEC}_{\Pi\text{-TS3-1},\mathcal{G}\text{-Setup},\mathcal{A},\mathcal{Z}}$ (see Protocol 2) This hybrid runs a modified protocol Π-TS3-1, which merges all Π-TS3$_{\mathcal{P}}$ (i.e. we group all Π-TS3$_{\mathcal{P}}$ into one machine similar to an ideal functionality) with Init and $\mathcal{P}$.Verify modified as follows:

- Init() now initializes an empty table VER as in $\mathcal{F}$-TS3.
- $\mathcal{P}$.Verify, after running $b :=_{\text{bool}} \mathsf{Verify}(\mathsf{pk}', m, \sigma)$, updates a table entry $\mathrm{VER}[\mathsf{pk}', m, \sigma]$ with b if it's not defined yet. Return $\mathrm{VER}[\mathsf{pk}', m, \sigma]$ instead of b.

Clearly, Hyb_1 is perfectly indistinguishable from Hyb_0.

$\mathsf{Hyb}_2 = \mathsf{EXEC}_{\Pi\text{-TS3-2},\mathcal{G}\text{-Setup},\mathcal{A},\mathcal{Z}}$ (see Protocol 3) This hybrid runs a modified protocol Π-TS3-2, which differs from the previous hybrid as follows:

Protocol 2: Hybrid Π-TS3-1

INIT()
1: $\mathsf{pk} := \bot$
2: $\mathrm{VER}[\cdot,\cdot,\cdot] := \bot$
3: $\mathrm{ST}_{\mathcal{P}}[\mathsf{eid}] := (0, \bot, \ldots, \bot)$ for all eid
4: $\mathrm{Ready}[\mathcal{P}] := 0$
5: $\mathrm{SGN}[\cdot,\cdot] := \varnothing$ // Never read (in this hybrid).

$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$
6: $b := \mathsf{Verify}(\mathsf{pk}', m, \sigma)$
7: **if** $\mathrm{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
8: $\quad \mathrm{VER}[\mathsf{pk}', m, \sigma] := b$
9: **return** $\mathrm{VER}[\mathsf{pk}', m, \sigma]$

– $\mathcal{P}$.VERIFY, after running $b{:=_{\text{bool}}}\mathsf{Verify}(\mathsf{pk}', m, \sigma)$, updates the verification bit b by running the function $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}', m)$ as in $\mathcal{F}$-TS3.
– $\mathcal{P}$.COMBINE$(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$, after running $\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}', \mathsf{trs})$, checks whether $\mathsf{ValidateC}_{\mathbf{t}}(\mathsf{pk}', \mathsf{trs})$ is true, and if so, runs $\sigma \leftarrow \mathsf{ForceValidity}(m, \sigma)$ as in $\mathcal{F}$-TS3.
– $\mathcal{P}$.COMBINE$(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$, after running $\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}', \mathsf{trs})$, checks whether $\mathsf{ValidateR}_{\mathbf{t}}(\mathsf{pk}', \mathsf{trs})$ is true, and if so, runs $\sigma \leftarrow \mathsf{ForceValidity}(m, \sigma)$ as in $\mathcal{F}$-TS3.

Consider the following three events.

– ErrorUF: During a $\mathcal{P}$.VERIFY call, $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}', m) = 1$ while $\mathsf{Verify}(\mathsf{pk}, m, \sigma) = 1$ (i.e., the signature σ is valid according to Verify but declared to be invalid).
– ErrorC: During a $\mathcal{P}$.COMBINE call, $\mathsf{ValidateC}_{\mathbf{t}}(\mathsf{pk}', \mathsf{trs}) = 1$ and $\mathsf{ForceValidity}(m, \sigma)$ detects that $\text{VERIFY}(\mathsf{pk}, m, \sigma) = 0$ (i.e., the signature σ is invalid according to VERIFY but declared to be valid).
– ErrorR: During a $\mathcal{P}$.COMBINE call, $\mathsf{ValidateR}_{\mathbf{t}}(\mathsf{pk}', \mathsf{trs}) = 1$ and $\mathsf{ForceValidity}(m, \sigma)$ detects that $\text{VERIFY}(\mathsf{pk}, m, \sigma) = 0$ (i.e., the signature σ is invalid according to VERIFY but declared to be valid).

If none of these events occur, then Hyb_2 behaves identically to Hyb_1. We show that the probability for any one of these events to occur is negligible.

Protocol 3: Hybrid Π-TS3-2

$\mathcal{P}$.COMBINE$(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$
1: $\mathsf{trs} := (\boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$
2: $\sigma \leftarrow \mathsf{Combine}(\mathsf{pk}', \mathsf{trs})$
3: **if** $\mathsf{ValidateC}_{\mathbf{t}}(\mathsf{pk}', \mathsf{trs})$ or $\mathsf{ValidateR}_{\mathbf{t}}(\mathsf{pk}', \mathsf{trs})$ **then**
4: $\quad \sigma \leftarrow \mathsf{ForceValidity}(m, \sigma)$
5: **return** σ

$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$
1: $b{:=_{\text{bool}}}\mathsf{Verify}(\mathsf{pk}', m, \sigma)$
2: **if** $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}', m)$ **then**
3: $\quad b := 0$
4: **if** $\mathrm{VER}[\mathsf{pk}', m, \sigma] = \bot$ **then**
5: $\quad \mathrm{VER}[\mathsf{pk}', m, \sigma] := b$
6: **return** $\mathrm{VER}[\mathsf{pk}', m, \sigma]$

Correctness error event We bound the probability that ErrorC is the first event among ErrorUF, ErrorC, and ErrorR to occur.

We can construct a reduction $\mathcal{B}$ winning the $\mathsf{Exp}^{\mathsf{cor},\mathsf{TCOR}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa,\mathbf{P},\mathbf{t})$ experiment if ErrorC occurs (before ErrorUF and ErrorR). The reduction $\mathcal{B}$ emulates Hyb_2 towards $\mathcal{Z}$ and $\mathcal{A}$, except the following differences:

1. On receiving par from the TCOR experiment, start running Hyb_2 towards $\mathcal{Z}$ and $\mathcal{A}$, receive the initial corrupted party information $\mathbf{C}$ from $\mathcal{A}$, and forward it to the TCOR experiment.
2. On receiving $(\mathsf{pk},(\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{C}})$ from the TCOR experiment, forward it to $\mathcal{A}$ and resume Hyb_2 towards $\mathcal{Z}$ and $\mathcal{A}$.
3. On receiving $\mathcal{F}$-Gen::$\mathcal{P}$.KeyGen() calls from corrupt $\mathcal{P}$, return the corresponding output $(\mathsf{pk},\mathsf{sk}_\mathcal{P})$.
4. On receiving a call to $\mathcal{P}$.KeyGen() from $\mathcal{Z}$ in Hyb_2, return pk.
5. On receiving a call to $\mathcal{P}$.Sign$_k$(eid, inp) from $\mathcal{Z}$ in Hyb_2, query $\mathcal{O}_{\text{Sign},k}$ with input $(\mathcal{P},\mathsf{eid},\mathsf{inp})$, and return the output to $\mathcal{Z}$.
6. On receiving Π-TS3$_\mathcal{P}$::$\mathcal{A}$.Corrupt() in Hyb_2, call $\mathcal{O}_{\text{Cor}}$ with input $\mathcal{P}$ and return the output to $\mathcal{A}$.
7. On receiving a call to $\mathcal{P}$.Combine(pk', $\boldsymbol{T}$, m, ctx, $(\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]}$) from $\mathcal{Z}$ in Hyb_2:
   - If ErrorC happens (and is the first event to happen), output $(\mathsf{pk}',\mathsf{trs})$ to the TCOR experiment, where $\mathsf{trs}=(\boldsymbol{T},m,ctx,(\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$.
   - Else, run $\mathcal{P}$.Combine as in Hyb_2 and return the output to $\mathcal{Z}$.

Clearly, $\mathcal{B}$ perfectly emulates Hyb_2 towards $\mathcal{Z}$ and $\mathcal{A}$ until an error event occurs. If the ErrorC event occurs (before ErrorUF, ErrorR), then $\mathsf{ValidateC}_\mathbf{t}(\mathsf{pk}',\mathsf{trs})=1$ and Verify$(\mathsf{pk},m,\sigma)=0$ for the signature σ output by Combine in Hyb_1. Note that because no error events happen before ErrorC, $\text{VER}[\mathsf{pk},m,\sigma']=\mathsf{Verify}(\mathsf{pk},m,\sigma')$ for all pk,m,σ'. Hence $\mathsf{ValidateC}_\mathbf{t}(\mathsf{pk}',\mathsf{trs})=1$ and $\mathsf{Verify}(\mathsf{pk},m,\sigma)=0$, satisfying the winning condition of the TCOR experiment.

Robustness error event We bound the probability that ErrorR is the first event among ErrorUF, ErrorC, and ErrorR to occur. We do this by reduction to the $\mathsf{Exp}^{\mathsf{cor},\mathsf{ROB}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa,\mathbf{P},\mathbf{t})$ experiment. The reduction is analogous to the previous reduction for ErrorC.

Unforgeability error event We bound the probability that ErrorUF is the first event among ErrorUF, ErrorC, and ErrorR to occur.

We can construct a reduction $\mathcal{B}$ winning the unforgeability experiment $\mathsf{Exp}^{\mathsf{cor},\mathsf{lvl},\mathsf{UF}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa,\mathbf{P},\mathbf{t})$. Similar to the previous reductions, the reduction $\mathcal{B}$ emulates Hyb_2 towards $\mathcal{Z}$ and $\mathcal{A}$. It is identical to the reduction for ErrorC until Step 6, except for the following difference:

7. On receiving a call to $\mathcal{P}$.Combine(pk', $\boldsymbol{T}$, m, ctx, $(\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]}$) from $\mathcal{Z}$ in Hyb_2, run $\mathcal{P}$.Combine as in Hyb_2 and return the output to $\mathcal{Z}$.
8. On receiving a call to $\mathcal{P}$.Verify(pk',σ,m) from $\mathcal{Z}$ in Hyb_2, if ErrorUF happens (and is the first event to happen), then:
   - If lvl = UF-X, output (m,σ) to the UF- experiment.
   - If lvl = SUF-X, collect $(\sigma_i)_{i\in[\ell]}$ such that each $\sigma_i\neq\sigma$ and $\text{VER}[\mathsf{pk},m,\sigma_i]=1$, and output $(m,\sigma,(\sigma_i)_{i\in[\ell]})$ to the SUF- experiment.

   Otherwise, run $\mathcal{P}$.Verify as in Hyb_2 and return the output to $\mathcal{Z}$.

Clearly, until an error event occurs, $\mathcal{B}$ perfectly emulates Hyb_2 towards $\mathcal{Z}$ and $\mathcal{A}$.

Note that because no error events happen before ErrorUF, $\text{VER}[\mathsf{pk},m,\sigma']=\mathsf{Verify}(\mathsf{pk},m,\sigma')$ for all pk,m,σ', hence $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}$ behaves the same between Hyb_2 and $\mathsf{Exp}^{\mathsf{cor},\mathsf{lvl},\mathsf{UF}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}$ (the bookkeeping VER is the same).

If lvl = UF-X, then $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}',m)=1$ and $\mathsf{Verify}(\mathsf{pk}',m,\sigma)=1$ implies $\mathcal{B}$ winning the unforgeability experiment. If lvl = SUF-X, then $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}',m)=1$ and $\mathsf{Verify}(\mathsf{pk}',m,\sigma)=1$ implies $\mathcal{B}$ winning the strong unforgeability experiment.

$\mathsf{Hyb}_3=\mathsf{EXEC}_{\Pi\text{-TS3-3},\mathcal{G}\text{-Setup},\mathcal{A},\mathcal{Z}}$ (see Protocol 4) This hybrid runs a modified protocol Π-TS3-3, which differs from the previous hybrid at Init and $\mathcal{P}$.KeyGen as follows:

- Init() asks the simulator $\mathcal{S}_{\text{TS3}}$ for values $(\mathsf{Verify},(\mathsf{Sign}_k)_{k=1}^{\mathsf{rnd}_{\text{sign}}},\mathsf{pk}',(\mathsf{st}_\mathcal{P},\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\boldsymbol{T}})$ as in $\mathcal{F}$-TS3.
- $\mathcal{P}$.KeyGen adds a check that the simulator did respond immediately (which $\mathcal{S}_{\text{TS3}}$ always does).

This modification is merely syntactic because $\mathcal{S}_{\text{TS3}}$ returns the same $\mathsf{Verify}, \mathsf{Sign}_k$ that Π-TS3 uses and runs the same procedure as $\mathcal{F}$-Gen::$\mathcal{P}$.KeyGen() to compute $\mathsf{pk}, \mathsf{st}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}}$. Thus, Hyb_3 and Hyb_2 are perfectly indistinguishable.

Finally, Hyb_3 is equivalent to $\mathsf{EXEC}_{\mathcal{F}\text{-TS3},\mathcal{G}\text{-Setup},\mathcal{S}_{\text{TS3}},\mathcal{Z}}$.

Protocol 4: Hybrid Π-TS3-3

Init()

1: $\mathsf{pk} := \bot$
2: $\text{VER}[\cdot,\cdot,\cdot] := \bot$
3: **for** $\mathcal{P} \in \mathbf{P}$ **do**
4: $\quad \text{ST}_{\mathcal{P}}[\mathsf{eid}] := (0, \bot, \ldots, \bot)$ for all eid
5: $\quad \text{Ready}[\mathcal{P}] := 0$
6: $\text{SGN}[\cdot,\cdot] := \varnothing$
7: $\mathsf{Sign}_k := (\cdot \mapsto (1,1,1))$ for $k \in [\mathsf{rnd}_{\text{sign}}]$ // Default
8: $\mathsf{Combine} := (\cdot \mapsto 1)$ // Default
9: $\mathsf{Verify} := (\cdot \mapsto 0)$ // Default
10: $\mathsf{st}_{\mathcal{P}} := (\mathcal{P}, \mathsf{pk}, \mathsf{sk}_{\mathcal{P}} := 1)$ for all $\mathcal{P} \in \mathbf{P}$
11: $(\mathsf{Verify}, (\mathsf{Sign}_k)_{k=1}^{\mathsf{rnd}_{\text{sign}}}, \mathsf{Combine}, \mathsf{pk}', (\mathsf{st}_{\mathcal{P}}, \mathsf{sk}_{\mathcal{P}})_{\mathcal{P}\in\mathbf{P}}) \leftarrow \mathcal{S}_{\text{TS3}}$::GetAlg()
12: // $\mathcal{S}_{\text{TS3}}$ always returns immediately (with honestly generated keys and the right algorithms), hence the defaults above will never be used and the if condition in the next line will always be true.
13: **if** $\mathsf{pk} = \bot$ **then**
14: $\quad \mathsf{pk} := \mathsf{pk}'$

$\mathcal{P}$.KeyGen() for $\mathcal{P} \in \mathbf{P}$

15: **if** $\mathsf{pk} = \bot$ **then** $\mathsf{pk} := 1$ // Will never be true for $\mathcal{S}_{\text{TS3}}$.
16: $\text{Ready}[\mathcal{P}] := 1$
17: **return** pk

Proof for "only if" direction. Suppose there exists an adversary $\mathcal{B}$ that breaks $\mathsf{Exp}^{\mathsf{cor},\mathsf{TCOR}}_{\Sigma_{\text{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t})$ with non-negligible probability. We will construct an environment $\mathcal{Z}$ and a UC adversary $\mathcal{A}$ such that that $\mathcal{Z}$ distinguishes $\mathsf{EXEC}_{\Pi\text{-TS3},\mathcal{G}\text{-Setup},\mathcal{A},\mathcal{Z}}$ from $\mathsf{EXEC}_{\mathcal{F}\text{-TS3},\mathcal{G}\text{-Setup},\mathcal{S},\mathcal{Z}}$ for any $\mathcal{S}$.

To simulate the view of $\mathcal{B}$ in the correctness game, we define $\mathcal{Z}$ as follows:

1. Call $\mathcal{P}$.Init() for all $\mathcal{P} \in \mathbf{P}$. If $\mathcal{Z}$ gets control back through $\mathcal{A}$, it says "ideal", since this indicates that the initialization was handed over to the simulator.
2. Call $\mathcal{G}$-Setup::$\mathcal{A}$.Setup() to receive par
3. Forward par to $\mathcal{B}$.
4. Upon receiving $\mathbf{C}$ from $\mathcal{B}$, instruct $\mathcal{A}$ to corrupt the parties in $\mathbf{C}$ and obtain $(\mathsf{sk}_{\mathcal{P}})_{\mathcal{P}\in\mathbf{C}}$.
5. Call $\mathcal{P}$.KeyGen() for all $\mathcal{P} \in \mathbf{P} \setminus \mathbf{C}$ and receives pk.
6. Forward $(\mathsf{pk}, (\mathsf{sk}_{\mathcal{P}})_{\mathcal{P}\in\mathbf{C}})$ to $\mathcal{B}$.
7. Upon receiving a query from $\mathcal{B}$ to $\mathcal{O}_{\text{Cor}}(\mathcal{P})$, instruct $\mathcal{A}$ to corrupt $\mathcal{P}$ and forward $(\mathsf{sk}_{\mathcal{P}}, \mathsf{st}_{\mathcal{P}}, \text{ST}_{\mathcal{P}})$ to $\mathcal{B}$.
8. Upon receiving a query from $\mathcal{B}$ to $\mathcal{O}_{\text{Sign},k}(\mathcal{P}, \mathsf{eid}, \mathsf{inp})$, for some $k \in \{1, \ldots, \mathsf{rnd}_{\text{sign}}\}$ and $\mathcal{P} \in \mathbf{P}$, call $\mathcal{P}.\text{Sign}_k(\mathsf{eid}, \mathsf{inp})$ and relay the response to $\mathcal{B}$.
9. Upon receiving $(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$ from $\mathcal{B}$, call

$$\mathcal{P}.\text{Combine}(\mathsf{pk}', \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell\in[\mathsf{rnd}_{\text{sign}}]})$$

   for some $\mathcal{P} \in \mathbf{P}$, to receive σ.
10. Call $\text{Verify}(\mathsf{pk}, m, \sigma)$ to obtain a decision bit b.
    If $\mathsf{Verify}(\mathsf{pk}, m, \sigma) = b = 0$ and $\mathsf{ValidateC}_{\mathbf{t}}(\mathsf{pk}, m, \sigma) = 1$, $\mathcal{Z}$ says "real"; otherwise, $\mathcal{Z}$ says "ideal".

Clearly, if $\mathcal{Z}$ is in the real world, then it outputs "real" with the same probability that $\mathcal{B}$ breaks the correctness experiment. If $\mathcal{Z}$ is in the ideal world, then it outputs "real" with probability 0, because if $\mathsf{ValidateC}_{\mathbf{t}} = 1$, then $\mathcal{F}$-TS3 forces σ to be valid according to VERIFY.

If there exists an adversary $\mathcal{B}$ that breaks $\mathsf{Exp}^{\mathsf{cor},\mathsf{ROB}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t})$ with non-negligible probability, we can construct an environment $\mathcal{Z}$ and a UC adversary $\mathcal{A}$ identically to the above.

Now we consider a case where there exists an adversary $\mathcal{B}$ that breaks $\mathsf{Exp}^{\mathsf{cor},\mathsf{lvl},\mathsf{UF}}_{\Sigma_{\mathsf{TS}},\mathcal{A}}(1^\kappa, \mathbf{P}, \mathbf{t})$ with non-negligible probability. Similar to the above, we will construct an environment $\mathcal{Z}$ and a UC adversary $\mathcal{A}$, simulating the view of $\mathcal{B}$ in the TUF game. $\mathcal{Z}$ behaves identically to the above until step 8 and then continues as follows:

9. Upon receiving $(m^*, \sigma^*, (\sigma_i^*)_{i\in[\ell]})$ from $\mathcal{B}$, call $\textsc{Verify}(\mathsf{pk}, m^*, \sigma_i^*)$ for all $i \in [\ell]$. $\mathcal{Z}$ then calls $\textsc{Verify}(\mathsf{pk}, m^*, \sigma^*)$ to obtain a decision bit b.
   If $\mathsf{Verify}(\mathsf{pk}, m^*, \sigma^*) = b = 1$ and $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}}(\mathsf{pk}, m^*) = 1$, $\mathcal{Z}$ says "real"; otherwise, $\mathcal{Z}$ says "ideal".

Clearly, if $\mathcal{Z}$ is in the real world, then it outputs "real" with the same probability that $\mathcal{B}$ breaks the unforgeability experiment. If $\mathcal{Z}$ is in the ideal world, then it outputs "real" with probability 0, because if $\mathsf{ValidateUF}_{\mathsf{lvl},\mathbf{t}} = 1$, then $\mathcal{F}$-TS3 forces σ to be invalid according to VERIFY. □

E Proof of Theorem 3 (Π-TS2 UC-realizes $\mathcal{F}$-TS2)

In this section, we provide the protocol Π-TS2 in Protocol 5. We prove the following theorem:

Theorem 3 (restated). *Let* $\mathsf{rnd}_{\mathrm{com}} = \mathsf{rnd}_{\mathrm{msg}} = \mathsf{rnd}_{\mathrm{T}} = \mathsf{rnd}_{\mathrm{ctx}} \in \{1, 2\}$. *Let* $t_{\mathrm{correct}} = t_{\mathrm{robust}} = |\mathbf{P}| + 1$ *(i.e., no correctness or robustness guarantees in* $\mathcal{F}$-TS3*). The protocol* Π-TS2$[\mathbf{P}, \mathbf{rnd}, \mathsf{cor}]$ *(uniform-)UC-realizes* $\mathcal{F}$-TS2$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}]$ *in the* $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ *and* $\mathcal{F}$-Net *hybrid model.*

Simulator 2: $\mathcal{S}_{\mathrm{TS2}}$

The simulator internally runs a copy of $\mathcal{A}$ and Π-TS2 (including its subfunctionalities $\mathcal{F}$-Net and $\mathcal{F}$-TS3).

$\mathcal{F}$-TS3::INIT() // Overwrite
1: $\mathcal{F}$-TS2.INIT()
2: // This will cause $\mathcal{S}_{\mathrm{TS2}}$::GETALG() to be called (if it has not been called yet).

$\mathcal{S}_{\mathrm{TS2}}$::GETALG()
3: $\mathcal{F}$-TS3::INIT() (Functionality 4)
   // This causes $\mathcal{F}$-TS3::INIT to query $\mathcal{A}$::GETALG(). Control is given to $\mathcal{A}$ (which may yield to the environment). After $\mathcal{A}$ returns (Verify, ...) to $\mathcal{F}$-TS3, execution of $\mathcal{F}$-TS3::INIT() concludes and control is handed back here.
4: $\mathcal{F}$-TS2::$\mathcal{S}$.KEYGEN() // Ensures pk from $\mathcal{F}$-TS3 is set in $\mathcal{F}$-TS2.
5: **return** $\mathcal{F}$-TS3::Verify

$\mathcal{S}_{\mathrm{TS2}}$::KEYGEN($\mathcal{P}$)
6: **if** $\mathcal{P} = \mathcal{S}_{\mathrm{TS2}}$ **then** pk := $\mathcal{F}$-TS3::pk
7: **else** pk ← Π-TS2$_{\mathcal{P}}$::$\mathcal{P}$.KEYGEN()
8: **return** pk

$\mathcal{S}_{\mathrm{TS2}}$::PREPROCESS($\mathcal{P}$, eid)
9: Run pp ← Π-TS2$_{\mathcal{P}}$::$\mathcal{P}$.PREPROCESS(eid)
10: **return** pp

(🕒)$\mathcal{S}_{\mathrm{TS2}}$::SIGN($\mathcal{P}$, $\boldsymbol{T}$, m, ctx, $(\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$)
11: $\sigma \leftarrow$ (🕒)Π-TS2$_{\mathcal{P}}$::$\mathcal{P}$.SIGN($\boldsymbol{T}$, m, ctx, $(\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q}\in\boldsymbol{T}}$)
12: **return** σ

Π-TS2$_{\mathcal{P}}$::$\mathcal{A}$.CORRUPT() // Overwrite
13: $\mathcal{F}$-TS2::$\mathcal{S}$.CORRUPT($\mathcal{P}$) // Corrupt in ideal world
14: **return** Π-TS2$_{\mathcal{P}}$::$\mathcal{A}$.CORRUPT() (Protocol 5)

$\mathcal{F}$-TS3::$\mathcal{A}$.CORRUPT($\mathcal{P}$) // Overwrite
15: $\mathcal{F}$-TS2::$\mathcal{S}$.CORRUPT($\mathcal{P}$) // Corrupt in ideal world
16: **return** $\mathcal{F}$-TS3::$\mathcal{A}$.CORRUPT($\mathcal{P}$) (Functionality 4)

Proof. We provide the final simulator $\mathcal{S}_{\text{TS2}}$ in Simulator 2. It is rather a trivial simulator, internally running a copy of the protocol Π-TS2. Whenever $\mathcal{S}_{\text{TS2}}$ runs a signing protocol step, it simply runs the corresponding step of Π-TS2 while internally running a copy of $\mathcal{F}$-TS3, $\mathcal{F}$-Net, and $\mathcal{A}$. Whenever Π-TS2 gives control to the adversary, $\mathcal{S}_{\text{TS2}}$ forwards the call to the internally simulated $\mathcal{A}$, waits for it to give control back, and then continues the simulation of Π-TS2.

This effectively represents the following setup:

- In the real world, preprocessing, signing and verifying are mostly governed by $\mathcal{F}$-TS3 (via Π-TS2).
- In the ideal world, preprocessing and signing are governed by $\mathcal{F}$-TS3 (via Π-TS2 via $\mathcal{S}_{\text{TS2}}$), while verification (and with it unforgeability assurance) is goverened by $\mathcal{F}$-TS2.

In this sense, the simulator runs its copy of $\mathcal{F}$-TS3 only for the purposes of preprocessing, signing, and to satisfy the adversary corrupting a party on $\mathcal{F}$-TS3. The main argument for the proof is about showing that verification in the real world ($\mathcal{F}$-TS3) behaves like the ideal world ($\mathcal{F}$-TS2).

To handle corruptions, the simulator simply executes the CORRUPT procedure of Π-TS2$_{\mathcal{P}}$ or $\mathcal{F}$-TS3 (which returns the protocol's state). Note that $\mathcal{S}_{\text{TS2}}$ has the full state of Π-TS2, $\mathcal{F}$-TS3, hence managing corruption is trivial. It also notifies $\mathcal{F}$-TS2 about the corruption to keep the $\mathbf{C}$ set in sync between real and ideal world.

A subtle aspect is initialization. Recall that when a functionality is first created (when it recceives its first call), its INIT procedure is executed. Special care needs to be taken for $\mathcal{F}$-TS3::INIT and $\mathcal{F}$-TS2::INIT. For this, consider the following two cases.

1. $\mathcal{F}$-TS3 is initialized when the environment makes its first query to $\mathcal{F}$-TS2 (or Π-TS2 in the real world). Then in the ideal world, this first query initializes $\mathcal{F}$-TS2, which calls $\mathcal{S}_{\text{TS2}}$::GETALG(), which initializes $\mathcal{F}$-TS3, which yields to the adversary $\mathcal{A}$::GETALG() (which may in turn yield to the environment). In the real world, the first query initializes Π-TS2$_{\mathcal{P}}$ (for some $\mathcal{P}$), which calls $\mathcal{S}_{\text{TS2}}$::GETALG() (as defined in Protocol 5) and hence causes the same chain of events as in the ideal world. Once $\mathcal{A}$ responds to its $\mathcal{A}$::GETALG call from $\mathcal{F}$-TS3, $\mathcal{S}_{\text{TS2}}$ also responds to its $\mathcal{S}_{\text{TS2}}$::GETALG call from $\mathcal{F}$-TS2 with the verification algorithm Verify that $\mathcal{A}$ returned. This means that at all times, $\mathcal{F}$-TS2 and Π-TS2 use the same verification algorithm.
2. The second case is that $\mathcal{F}$-TS3 is initialized before the environment makes its first query. This can happen if (and only if) $\mathcal{A}$ calls a procedure on $\mathcal{F}$-TS3 before the environment makes any calls to $\mathcal{F}$-TS2/Π-TS2. In this case, in the real world, $\mathcal{A}$'s call initializes $\mathcal{F}$-TS3, which causes a $\mathcal{A}$::GETALG() call (which allows the adversary to yield to the environment). In the ideal world, because $\mathcal{S}_{\text{TS2}}$ overwrites $\mathcal{F}$-TS3::INIT, the call of $\mathcal{A}$ that initializes $\mathcal{F}$-TS3 instead causes $\mathcal{F}$-TS2::INIT(). This causes a $\mathcal{S}_{\text{TS2}}$::GETALG call, which then causes the original $\mathcal{F}$-TS3::INIT to be called, which causes $\mathcal{A}$::GETALG() as in the real world. Again, once $\mathcal{A}$ responds to GETALG, so does $\mathcal{S}_{\text{TS2}}$, and $\mathcal{F}$-TS2 and Π-TS2 use the same verification algorithm at all times.

Apart from initialization minutiae, calls to $\mathcal{F}$-TS2 generally just cause the simulator to run the corresponding Π-TS2 procedures verbatim, just like in the real world. The details are straightforward to check.

The exception to this are calls to VERIFY, which are answered by $\mathcal{F}$-TS2 and Π-TS2 (via $\mathcal{F}$-TS3) in exactly the same way. This requires some argumentation. First, the code of $\mathcal{F}$-TS2::VERIFY and $\mathcal{F}$-TS3::VERIFY is identical and the algorithm Verify is also the same for both at all points in time, as we discussed for initialization. Second, pk is the same for both ideal world $\mathcal{F}$-TS2 and real world $\mathcal{F}$-TS3 at all points in time by design of $\mathcal{S}_{\text{TS2}}$: Either $\mathcal{F}$-TS3::pk is set after $\mathcal{A}$ responds to GETALG, then $\mathcal{S}_{\text{TS2}}$ also sets $\mathcal{F}$-TS2::pk in Line 4, or $\mathcal{F}$-TS3::pk is set to 1 during a call to $\mathcal{F}$-TS3::$\mathcal{P}$.KEYGEN, in which case $\mathcal{F}$-TS2::pk is set right after. Third, the SGN data structure is the same between ideal world ($\mathcal{F}$-TS2) and ideal world ($\mathcal{F}$-TS3): for both $\mathsf{rnd}_{\text{com}} \in \{1, 2\}$, the protocol Π-TS2 causes $\mathcal{F}$-TS3 to add $\mathcal{P}$ to SGN$[m, ssid]$ at the start of Π-TS2::SIGN. In $\mathcal{F}$-TS2, $\mathcal{P}$ is similarly added to SGN$[m, ssid]$ at the start of $\mathcal{F}$-TS2::SIGN. Fourth, the VER data structure is the same between ideal world $\mathcal{F}$-TS2 and real world $\mathcal{F}$-TS3. It is only written to in VERIFY for $\mathcal{F}$-TS2. In $\mathcal{F}$-TS3 in the real world, it is written to in the same way in VERIFY. In principle, $\mathcal{F}$-TS3 could also write to VER through ForceValidity. However, because $t_{\text{correct}} = t_{\text{robust}} = |\mathbf{P}|+1$ (i.e., correctness and robustness are disabled), ForceValidity is never called in $\mathcal{F}$-TS3. Overall, the VERIFY procedures of $\mathcal{F}$-TS2 and $\mathcal{F}$-TS3 work on effectively the same data structure and behave identically. □

Protocol 5: Π-TS2$_{\mathcal{P}}[\mathbf{P}, \mathbf{rnd}, \mathsf{cor}]$ realizing $\mathcal{F}$-TS2

With access to $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ (Functionality 4) and $\mathcal{F}$-Net (Functionality 1). We require that $\mathsf{rnd}_{\mathrm{com}} = \mathsf{rnd}_{\mathrm{msg}} = \mathsf{rnd}_{\mathrm{T}} = \mathsf{rnd}_{\mathrm{ctx}} \in \{1, 2\}$.

(⏲)INIT()
1: $\mathrm{PM}_{\mathcal{P}}[\mathsf{eid}] = (), \mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] := \bot$ for all $\mathsf{eid} \in \{0,1\}^*$
2: (⏲)$\mathcal{F}$-TS3.INIT() // Only runs if $\mathcal{F}$-TS3 has not been initialized before.

$\mathcal{P}$.KEYGEN() if $\mathcal{P} \in \mathbf{P}$
3: **return** $\mathsf{pk} \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.KEYGEN()

$\mathcal{P}$.PREPROCESS(eid) if $\mathcal{P} \in \mathbf{P}$
4: **assert** $\mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] = \bot$
5: **if** $\mathsf{rnd}_{\mathrm{com}} = 1$ **then** // No preprocessing.
6: **return** $\mathsf{pm}' := \bot$
7: **else**
8: $\mathsf{inp} := ()$ // Empty input.
9: $\mathsf{pm}' \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_1$(eid, inp)
10: $\mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] := \mathsf{pm}'$
11: **return** $\mathsf{pp} = \mathsf{pm}'$

(⏲)$\mathcal{P}$.SIGN$(\boldsymbol{T}, m, ctx, (\mathsf{eid}_{\mathcal{Q}}, \mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$ if $\mathcal{P} \in \mathbf{P}$
12: **assert** $\mathcal{P} \in \boldsymbol{T}$
13: $\mathsf{eid} := \mathsf{eid}_{\mathcal{P}}$
14: **assert** $\mathrm{PP}_{\mathcal{P}}[\mathsf{eid}] = \mathsf{pp}_{\mathcal{P}} \neq \bot$
15: **if** $\mathsf{rnd}_{\mathrm{com}} = 1$ **then** // No preprocessing. Need to run first round.
16: (⏲)RunRound$_1(\mathcal{P}, \mathsf{eid}, \boldsymbol{T}, m, ctx)$
17: **else**// First round has happened during preprocessing. Store messages.
18: $(\mathsf{eid}, \mathsf{pp}_{\mathcal{P}}) := \mathrm{PP}_{\mathcal{P}}[\mathsf{eid}]$
19: $\mathbf{pm}_1 := (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$ // Preprocessing messages from input.
20: $\mathrm{PM}_{\mathcal{P}}[\mathsf{eid}] := (\mathbf{pm}_1)$
21: **for** $k = 2, \ldots, \mathsf{rnd}_{\mathrm{sign}}$ **do** // Run remaining rounds.
22: (⏲)RunRound$_k(\mathcal{P}, \mathsf{eid}, \boldsymbol{T}, m, ctx, (\mathsf{eid}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$
23: $(\mathbf{pm}_\ell)_{\ell \in [\mathsf{rnd}_{\mathrm{sign}}]} := \mathrm{PM}_{\mathcal{P}}[\mathsf{eid}]$
24: $\sigma \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.COMBINE$(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [\mathsf{rnd}_{\mathrm{sign}}]})$
25: **assert** $\mathcal{P}$.VERIFY$(\mathsf{pk}, m, \sigma) = 1$
26: **return** σ

$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$
27: **return** $\mathcal{F}$-TS3::$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$

(⏲)RunRound$_k(\mathcal{P}, \mathsf{eid}, \boldsymbol{T}, m, ctx, (\mathsf{eid}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$ for $k \geq 1$ (internal)
28: $(\mathbf{pm}_\ell)_{\ell \in [k-1]} := \mathrm{PM}_{\mathcal{P}}[\mathsf{eid}]$
29: $\mathsf{inp}.\mathrm{T} := \boldsymbol{T}, \mathsf{inp}.\mathrm{m} := m, \mathsf{inp}.\mathrm{ctx} := ctx$
30: **if** $k > 1$ **then** $\mathsf{inp}.\mathbf{pm} := \mathbf{pm}_{k-1}$
31: $\mathsf{pm}'_{\mathcal{P}} \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_k(\mathsf{eid}, \mathsf{inp})$ // $\mathcal{F}$-TS3 enforces that eid is only used once, etc.
32: **for** $\mathcal{Q} \in \boldsymbol{T}$ **do**
33: (⏲)$\mathcal{F}$-Net::$\mathcal{P}$.SEND$(\mathcal{Q}, port := \mathsf{eid}_{\mathcal{Q}}, (\boldsymbol{T}, m, \mathsf{pm}'_{\mathcal{P}}))$ // Send message $\mathsf{pm}'_{\mathcal{P}}$ to all other signers.
34: **for** $\mathcal{Q} \in \boldsymbol{T}$ **do** $(\cdot, \cdot, \mathsf{pm}'_{\mathcal{Q}}) \leftarrow$ (⏲)$\mathcal{F}$-Net::$\mathcal{P}$.RECEIVE$(\mathcal{Q}, port := \mathsf{eid})$ // Receive messages from all other signers.
35: $\mathbf{pm}_k := (\mathsf{pm}'_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$
36: $\mathrm{PM}_{\mathcal{P}}[\mathsf{eid}] := (\mathbf{pm}_\ell)_{\ell \in [k]}$

$\mathcal{A}$.CORRUPT()
37: **assert** $\mathcal{P} \in \mathbf{P}$
38: **assert** $\mathsf{cor} =$ adaptive $\vee$ [$\mathcal{P}$.KEYGEN has not been called yet]
39: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}, \mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
40: **return** $(\mathrm{PP}_{\mathcal{P}}, \mathrm{PM}_{\mathcal{P}})$

Protocol 6: Π-TS1$_{\mathcal{P}}[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ realizing $\mathcal{F}$-TS1

With access to $\mathcal{F}$-TS3 (Functionality 4) and $\mathcal{F}$-Net (Functionality 1). We require $\mathsf{rnd}_{\mathrm{com}} = \mathsf{rnd}_{\mathrm{msg}} = \mathsf{rnd}_{\mathrm{T}} = \mathsf{rnd}_{\mathrm{ctx}} = 1$.

INIT()
1: $\mathrm{PM}_{\mathcal{P}}[\mathsf{eid}] = ()$ for all $\mathsf{eid} \in \{0,1\}^*$
2: (🕒)$\mathcal{F}$-TS3.INIT()

$\mathcal{P}$.KEYGEN() for $\mathcal{P} \in \mathbf{P}$
3: **return** $\mathsf{pk} \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.KEYGEN()

(🕒)$\mathcal{P}$.SIGN$(\boldsymbol{T}, m)$ for $\mathcal{P} \in \mathbf{P}$
4: **assert** $\mathcal{P} \in \boldsymbol{T}$
5: $\mathsf{eid} \xleftarrow{\$} \{0,1\}^{2\kappa}$
6: **for** $k = 1, \ldots, \mathsf{rnd}_{\mathrm{sign}}$ **do** // Run rounds.
7: (🕒)$\mathsf{RunRound}_k(\mathcal{P}, \mathsf{eid}, \boldsymbol{T}, m)$
8: $(\mathbf{pm}_\ell)_{\ell \in [\mathsf{rnd}_{\mathrm{sign}}]} := \mathrm{PM}_{\mathcal{P}}[\mathsf{eid}]$
9: $\sigma \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.COMBINE$(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_\ell)_{\ell \in [\mathsf{rnd}_{\mathrm{sign}}]})$
10: **assert** $\mathcal{P}$.VERIFY$(\mathsf{pk}, m, \sigma) = 1$
11: **return** σ

$\mathcal{P}$.VERIFY(pk, m, σ) for all $\mathcal{P}$
12: **return** $\mathcal{F}$-TS3.VERIFY(pk, m, σ)

(🕒)$\mathsf{RunRound}_k(\mathcal{P}, \mathsf{eid}, \boldsymbol{T}, m)$ for $k \geq 1$ (internal)
13: $(\mathbf{pm}_\ell)_{\ell \in [k-1]} := \mathrm{PM}_{\mathcal{P}}[\mathsf{eid}]$
14: $\mathsf{inp.T} := \boldsymbol{T}, \mathsf{inp.m} := m, \mathsf{inp.ctx} := 0$
15: **if** $k > 1$ **then** $\mathsf{inp.pm} := \mathbf{pm}_{k-1}$
16: $\mathsf{pm}'_{\mathcal{P}} \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.$\mathrm{SIGN}_k(\mathsf{eid}, \mathsf{inp})$
17: **for** $\mathcal{Q} \in \boldsymbol{T}$ **do** (🕒)$\mathcal{F}$-Net::$\mathcal{P}$.SEND$(\mathcal{Q}, port := (\boldsymbol{T}, m, k), \mathsf{pm}'_{\mathcal{P}})$ // Send message $\mathsf{pm}'_{\mathcal{P}}$ to all signers.
18: **for** $\mathcal{Q} \in \boldsymbol{T}$ **do** $\mathsf{pm}'_{\mathcal{Q}} \leftarrow$ (🕒)$\mathcal{F}$-Net::$\mathcal{P}$.RECEIVE$(\mathcal{Q}, port := (\boldsymbol{T}, m, k))$ // Receive messages from other signers.
19: $\mathbf{pm}_k := (\mathsf{pm}'_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$
20: $\mathrm{PM}_{\mathcal{P}}[\mathsf{eid}] := (\mathbf{pm}_\ell)_{\ell \in [k]}$

$\mathcal{A}$.CORRUPT()
21: **assert** $\mathcal{P} \in \mathbf{P}$
22: **assert** $\mathsf{cor} =$ adaptive $\vee$ [$\mathcal{P}$.KEYGEN has not been called yet]
23: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}, \mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
24: **return** $\mathrm{PM}_{\mathcal{P}}$ // The only state of Π-TS1$_{\mathcal{P}}$

F Proof of Theorem 4 (Π-TS1 UC-realizes $\mathcal{F}$-TS1)

In this section, we prove the following theorem about Π-TS1 (Protocol 6).

Theorem 4 (restated). *Let* $\mathsf{lvl} =$ UF-1. *Let* $\mathsf{rnd}_{\mathrm{com}} = \mathsf{rnd}_{\mathrm{msg}} = \mathsf{rnd}_{\mathrm{T}} = \mathsf{rnd}_{\mathrm{ctx}} = 1$. *Let* $t_{\mathrm{correct}} = t_{\mathrm{robust}} = |\mathbf{P}|+1$ *(i.e., no correctness or robustness guarantees in* $\mathcal{F}$-TS3*). The protocol* Π-TS1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ *(uniform-)UC-realizes* $\mathcal{F}$-TS1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ *in the* $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}]$ *and* $\mathcal{F}$-Net *hybrid model.*

Proof. The proof is analogous to that of Theorem 3. We provide the simulator $\mathcal{S}_{\mathrm{TS1}}$ in Simulator 3. The main difference is that $\mathcal{S}_{\mathrm{TS1}}$ does not need to handle a preprocessing step, as Π-TS1 does not have one. Rather, it just forwards the signing command to the internally simulated Π-TS1.

Calls to VERIFY are answered by $\mathcal{F}$-TS1 and Π-TS1 (via $\mathcal{F}$-TS3) in exactly the same way. For this, note that $\mathcal{F}$-TS1::SGN$[m] = \bigcup_{ssid} \mathcal{F}$-TS3::SGN$[m, ssid]$. With this correspondence, the VERIFY procedures of $\mathcal{F}$-TS1 and $\mathcal{F}$-TS3 work on effectively the same data (SGN, VER) and hence behave identically. The argument for VER, pk being synchronized between ideal $\mathcal{F}$-TS1 and real $\mathcal{F}$-TS3 is the same as in the proof of Theorem 3. □

Simulator 3: $\mathcal{S}_{\text{TS1}}$

The simulator internally runs a copy of $\mathcal{A}$ and Π-TS1 (including its subfunctionalities $\mathcal{F}$-Net and $\mathcal{F}$-TS3).

$\mathcal{F}$-TS3::Init() // Overwrite
1: $\mathcal{F}$-TS1.Init()
2: // This will cause $\mathcal{S}_{\text{TS1}}$::GetAlg() to be called (if it has not been called yet).

$\mathcal{S}_{\text{TS1}}$::GetAlg()
3: $\mathcal{F}$-TS3::Init() (Functionality 4)
// This causes $\mathcal{F}$-TS3::Init to query $\mathcal{A}$::GetAlg(). Control is given to $\mathcal{A}$ (which may yield to the environment). After $\mathcal{A}$ returns (Verify, ...) to $\mathcal{F}$-TS3, execution of $\mathcal{F}$-TS3::Init() concludes and control is handed back here.
4: $\mathcal{F}$-TS1::$\mathcal{S}$.KeyGen() // Ensures pk from $\mathcal{F}$-TS3 is set in $\mathcal{F}$-TS1.
5: **return** $\mathcal{F}$-TS3::Verify

$\mathcal{S}_{\text{TS1}}$::KeyGen($\mathcal{P}$)
6: **if** $\mathcal{P} = \mathcal{S}_{\text{TS2}}$ **then** pk := $\mathcal{F}$-TS3::pk
7: **else** pk $\leftarrow$ Π-TS1$_{\mathcal{P}}$::$\mathcal{P}$.KeyGen()
8: **return** pk

(⏲)$\mathcal{S}_{\text{TS1}}$::Sign($\mathcal{P}$, $\boldsymbol{T}$, m)
9: **return** $\sigma \leftarrow$ (⏲)Π-TS1$_{\mathcal{P}}$::$\mathcal{P}$.Sign($\boldsymbol{T}, m$)

Π-TS1$_{\mathcal{P}}$::$\mathcal{A}$.Corrupt() // Overwrite
10: $\mathcal{F}$-TS1::$\mathcal{S}$.Corrupt($\mathcal{P}$) // Corrupt in ideal world
11: **return** Π-TS1$_{\mathcal{P}}$::$\mathcal{A}$.Corrupt() (Protocol 6)

$\mathcal{F}$-TS3::$\mathcal{A}$.Corrupt($\mathcal{P}$) // Overwrite
12: $\mathcal{F}$-TS1::$\mathcal{S}$.Corrupt($\mathcal{P}$) // Corrupt in ideal world
13: **return** $\mathcal{F}$-TS3::$\mathcal{A}$.Corrupt($\mathcal{P}$) (Functionality 4)

G Π-TSSync2 and Proof of Theorem 5 (Π-TSSync2 UC-realizes $\mathcal{F}$-TSSync2)

In the remainder of this section, we present the protocol Π-TSSync2 (Protocol 7) that UC-realizes the ideal functionality $\mathcal{F}$-TSSync2 (Functionality 9) in the $\mathcal{G}$-Clock and $\mathcal{F}$-NetSync (Functionality 8) hybrid model.

The protocol is straightforward: Preprocessing is delegated to $\mathcal{F}$-TS3. The Sign procedure queues a new signing process (but does not run any computation yet other than bookkeeping). The actual signing happens in Tick, which receives protocol messages, executes the next round of all ongoing signing processes and sends the resulting protocol messages via $\mathcal{F}$-NetSync. During GetSig, the protocol invoked $\mathcal{F}$-TS3::Combine to compute the final signature. Intuitively, unforgeability is guaranteed by $\mathcal{F}$-TS3::Verify, and correctness and robustness are guaranteed $\mathcal{F}$-TS3::Combine.

Theorem 5 (restated). The protocol Π-TSSync2[$\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}$] (uniform-)UC-realizes the ideal functionality $\mathcal{F}$-TSSync2[$\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}$] in the $\mathcal{F}$-NetSync and $\mathcal{F}$-TS3[$\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd} = (\mathsf{rnd}_{\text{T}} = 2, \mathsf{rnd}_{\text{msg}} = 2, \mathsf{rnd}_{\text{ctx}} = 2, \mathsf{rnd}_{\text{com}} = 2, \mathsf{rnd}_{\text{sign}}), \mathsf{ssid}, \mathsf{cor}$] hybrid model in the presence of $\mathcal{G}$-Clock.

Proof. Let $\mathcal{S}_{\text{TSSync2}}$ be the simulator (Simulator 4) that internally runs a copy of $\mathcal{A}$ and Π-TSSync2 (including its subfunctionalities $\mathcal{F}$-NetSync and $\mathcal{F}$-TS3). It has access to (but does not control) $\mathcal{G}$-Clock. With this simulator, which mostly runs the honest protocol, it is easy to check that the ideal world with $\mathcal{F}$-TSSync2 and $\mathcal{S}_{\text{TSSync2}}$ behaves exactly like the real world with Π-TSSync2, $\mathcal{F}$-TS3, $\mathcal{F}$-NetSync, and $\mathcal{A}$.

The setup works as follows:

- Preprocessing and generating signing protocol messages every round are governed by $\mathcal{F}$-TS3 in both the real world (via Π-TSSync2) and ideal world (via the copy of $\mathcal{F}$-TS3 in $\mathcal{S}_{\text{TSSync2}}$).
- Verification of signatures and enforcement of unforgeability are done by $\mathcal{F}$-TS3 in the real world (via Π-TSSync2) and by $\mathcal{F}$-TSSync2 in the ideal world.
- Combining the resulting signature during GetSig happens via the Combine algorithm (output by $\mathcal{A}$ during initialization) in both real and ideal world. Enforcing correctness and robustness of

Protocol 7: Π-TSSync2$_{\mathcal{P}}[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}]$ realizing $\mathcal{F}$-TSSync2

with access to $\mathcal{G}$-Clock and $\mathcal{F}$-NetSync and $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd} = (\mathsf{rnd}_{\text{T}} = 2, \mathsf{rnd}_{\text{msg}} = 2, \mathsf{rnd}_{\text{ctx}} = 2, \mathsf{rnd}_{\text{com}} = 2, \mathsf{rnd}_{\text{sign}}), \mathsf{ssid}, \mathsf{cor}]$.

INIT()
1: $\text{Proc}[\cdot] := \varnothing$
2: $\text{PP}_{\mathcal{P}}[\cdot] := \bot$ for all $\mathcal{P} \in \mathbf{P}$
3: $\text{PM}_{\mathcal{P}}[\cdot] := ()$ for all $\mathcal{P} \in \mathbf{P}$
4: $\text{Used}_{\mathcal{P}} = \varnothing$
5: $\text{RndDone}[\cdot] := 0$
6: $\mathsf{pk} := \bot$
7: $\text{EID}_{\mathcal{P}}[\cdot] := \bot$
8: (⏲)$\mathcal{F}$-TS3.INIT()

$\mathcal{P}$.KEYGEN() if $\mathcal{P} \in \mathbf{P}$
9: **return** $\mathsf{pk} :=$ $\mathcal{F}$-TS3::$\mathcal{P}$.KEYGEN()

$\mathcal{A}$.CORRUPT()
10: **assert** $\mathcal{P} \in \mathbf{P}$
11: **assert** cor = adaptive $\vee$ [$\mathcal{P}$.KEYGEN has not been called yet]
12: $\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
13: **return** $(\text{Proc}, \text{PP}_{\mathcal{P}}, \text{PM}_{\mathcal{P}}, \text{Used}_{\mathcal{P}}, \text{RndDone}, \mathsf{pk}, \text{EID}_{\mathcal{P}})$

$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$
14: **return** $\mathcal{F}$-TS3::$\mathcal{P}$.VERIFY$(\mathsf{pk}', m, \sigma)$

$\mathcal{P}$.PREPROCESS(eid) if $\mathcal{P} \in \mathbf{P}$
15: **assert** $\mathcal{P}$.KEYGEN() has been called.
16: $\mathsf{pp} \leftarrow$ $\mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_1(\mathsf{eid}, \mathsf{inp} := ())$
17: $\text{PP}_{\mathcal{P}}[\mathsf{eid}] := \mathsf{pp}$
18: **return** pp

$\mathcal{P}$.SIGN$(\mathsf{eid}, \boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$ if $\mathcal{P} \in \mathbf{P}$
19: $\mathcal{G}$-Clock::$\mathcal{P}$.REGISTER() // Ensure honest parties wait on $\mathcal{P}$ to finish round.
20: **assert** $\boldsymbol{T} \subseteq \mathbf{P} \wedge \mathcal{P} \in \boldsymbol{T}$
21: **assert** $\text{PP}_{\mathcal{P}}[\mathsf{eid}] = \mathsf{pp}_{\mathcal{P}} \neq \bot$
22: **assert** $\mathsf{eid} \notin \text{Used}_{\mathcal{P}}$
23: $\text{Used}_{\mathcal{P}} := \text{Used}_{\mathcal{P}} \cup \{\mathsf{eid}\}$
24: $ssid := \mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}})$
25: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GETROUND()
26: **assert** $now >$ RndDone$[\mathcal{P}]$ // $\mathcal{P}$.TICK() not yet called this round.
27: $round_{\text{start}} := now$ // Start this round.
28: $proc := (\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}})$
29: $\text{Proc}[proc] := \text{Proc}[proc] \cup \{\mathcal{P}\}$
30: $\text{EID}_{\mathcal{P}}[proc] := \mathsf{eid}$
31: $\text{PM}_{\mathcal{P}}[\mathsf{eid}] := \mathbf{pm}_0 := (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}$
32: $\mathsf{eid}' \xleftarrow{\$} \{0,1\}^{2\kappa}$
33: $\mathsf{inp} := (\boldsymbol{T}, m, ctx, \text{PM}_{\mathcal{P}}[\mathsf{eid}])$
34: $\mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_2(\mathsf{eid}', \mathsf{inp})$ // Sole purpose: make $\mathcal{F}$-TS3 add $\mathcal{P}$ to SGN$[m, ssid]$ already.

$\mathcal{P}$.GETSIG$(proc)$ if $\mathcal{P} \in \mathbf{P}$
35: ReadRound$(\mathcal{P})$ // Read latest network msgs
36: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GETROUND()
37: $(\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}}) := proc$
38: **assert** $\mathcal{P} \in \text{Proc}[proc]$ // $\mathcal{P}$ participated.
39: **assert** $now \geq round_{\text{start}} + \mathsf{rnd}_{\text{sign}} - 1$ // Finished
40: $\mathsf{eid} := \text{EID}_{\mathcal{P}}[proc]$
41: $\sigma \leftarrow$ $\mathcal{F}$-TS3::$\mathcal{P}$.COMBINE$(\mathsf{pk}, \boldsymbol{T}, m, ctx, \text{PM}_{\mathcal{P}}[\mathsf{eid}])$
42: **return** σ

(⏲)$\mathcal{P}$.TICK() if $\mathcal{P} \in \mathbf{P}$ // Send messages and finish round
43: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GETROUND()
44: **assert** $\text{RndDone}[\mathcal{P}] < now$
45: $\text{RndDone}[\mathcal{P}] := now$
46: ReadRound$(\mathcal{P})$ // Read latest messages from network, update finished signatures
47: **for** $proc := (\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}})$ with $\mathcal{P} \in \text{Proc}[proc]$ **do**
48: $\mathsf{eid} := \text{EID}_{\mathcal{P}}[proc]$
49: $round := now - round_{\text{start}} + 2$ // Sign_{round} to be called in this round.
50: **if** $round \leq \mathsf{rnd}_{\text{sign}}$ **then** // Still rounds to execute.
51: $\mathsf{inp} := (\boldsymbol{T}, m, ctx, \text{PM}_{\mathcal{P}}[\mathsf{eid}])$
52: $\mathsf{pm}' \leftarrow$ $\mathcal{F}$-TS3::$\mathcal{P}$.SIGN$_{round}(\mathsf{eid}, \mathsf{inp})$
53: **for** $\mathcal{R} \in \boldsymbol{T}$ **do**
54: $\mathcal{F}$-NetSync::$\mathcal{P}$.SEND$(\mathcal{R}, port := proc, \mathsf{pm}')$
55: (⏲)$\mathcal{G}$-Clock::$\mathcal{P}$.ROUNDOK()

ReadRound$(\mathcal{P})$ (internal) // Receives messages and updates internal data structures. Called at least once every round for each honest user (by TICK). Safe to call multiple times per round.
56: $now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GETROUND()
57: **for** $proc := (\boldsymbol{T}, m, ctx, (\mathsf{pp}_{\mathcal{Q}})_{\mathcal{Q} \in \boldsymbol{T}}, ssid, round_{\text{start}})$ with $\mathcal{P} \in \text{Proc}[proc]$ **do**
58: $\mathsf{eid} := \text{EID}_{\mathcal{P}}[proc]$
59: $round := now - round_{\text{start}} + 2$ // Current protocol round (receiving messages from $round - 1$)
60: **if** $round < 3 \vee round > \mathsf{rnd}_{\text{sign}} + 1$ **then** skip below and continue for loop with next $proc$
61: $(\mathbf{pm}_k)_{k=1}^{\ell} := \text{PM}_{\mathcal{P}}[\mathsf{eid}]$ // Read previously received messages. $\ell \in \{round - 2, round - 1\}$
62: $\mathbf{pm}_{round-1} :=$ $\mathcal{F}$-NetSync::$\mathcal{P}$.RECEIVE$(port := proc)$
63: $\mathbf{pm}_{round-1} := (\mathbf{pm}_{round-1}[\mathcal{Q}])_{\mathcal{Q} \in \boldsymbol{T}}$ // Filter $\mathbf{pm}_{round-1}$ for $\boldsymbol{T}$.
64: $\text{PM}_{\mathcal{P}}[\mathsf{eid}] := (\mathbf{pm}_k)_{k=1}^{round-1}$ // Update $\text{PM}_{\mathcal{P}}[\mathsf{eid}]$ with $\mathbf{pm}_{round-1}$

that signature (calling ForceValidity) is done by $\mathcal{F}$-TS3 in the real world and by $\mathcal{F}$-TSSync2 in the ideal world.

Simulator 4: $\mathcal{S}_{\text{TSSync2}}$

The simulator internally runs a copy of $\mathcal{A}$ and Π-TSSync2 (including its subfunctionalities $\mathcal{F}$-NetSync and $\mathcal{F}$-TS3). It has access to (but does not control) $\mathcal{F}$-TSSync2 and to $\mathcal{G}$-Clock.

Π-TS2$_\mathcal{P}$::$\mathcal{A}$.Corrupt() // Overwrite

1: $\mathcal{F}$-TSSync2::$\mathcal{S}$.Corrupt($\mathcal{P}$) // Corrupt in ideal world
2: SyncState($\mathcal{P}$) // Catch up on missed calls
3: **return** Π-TSSync2$_\mathcal{P}$::$\mathcal{A}$.Corrupt() (Protocol 7)

$\mathcal{F}$-TS3::$\mathcal{A}$.Corrupt($\mathcal{P}$) // Overwrite

4: $\mathcal{F}$-TSSync2::$\mathcal{S}$.Corrupt($\mathcal{P}$) // Corrupt in ideal world
5: **return** $\mathcal{F}$-TS3::$\mathcal{A}$.Corrupt($\mathcal{P}$) (Functionality 4)

$\mathcal{F}$-TS3::Init() // Overwrite

6: $\mathcal{F}$-TSSync2.Init()
// Causes $\mathcal{S}_{\text{TSSync2}}$::GetAlg().

$\mathcal{S}_{\text{TSSync2}}$::GetAlg()

7: $\mathcal{F}$-TS3::Init() (Functionality 4)
8: // This causes $\mathcal{F}$-TS3::Init to query $\mathcal{A}$::GetAlg(). Control is given to $\mathcal{A}$ (which may yield to the environment). After $\mathcal{A}$ returns (Verify, ..., pk′, ...) to $\mathcal{F}$-TS3, execution of $\mathcal{F}$-TS3::Init() concludes and control is handed back here.
9: Define function $\mathsf{Preprocess}_{\text{FTS3}}(\mathcal{P}, \mathsf{eid}, st_{\text{pp}})$ to interpret st_{pp} as the current state of $\mathcal{F}$-TS3, execute $\mathsf{pp} \leftarrow \mathcal{F}$-TS3::$\mathcal{P}$.Sign$_1(\mathsf{eid}, \mathsf{inp} := ())$,update st_{pp} according to the changed $\mathcal{F}$-TS3 state, and return $\mathsf{pp}, st_{\text{pp}}$
10: **return** $\mathcal{F}$-TS3::Verify, $\mathsf{Preprocess}_{\text{FTS3}}$, $\mathcal{F}$-TS3::pk

SyncState($\mathcal{P}$) // Internal helper. Synchronizes missed Sign calls from $\mathcal{F}$-TSSync2 to Π-TSSync2

11: // Take $\mathcal{F}$-TSSync2::$\mathcal{P}$.Sign calls we missed (since $\mathcal{S}_{\text{TSSync2}}$ is not invoked for Sign calls) and update Π-TSSync2 state with them.
12: $(\text{Proc}, (\text{EID}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}, st_{\text{pp}}) := \mathcal{F}$-TSSync2::$\mathcal{S}_{\text{TSSync2}}$.GetProcessInfo() // Info about missing calls.
13: Π-TSSync2$_\mathcal{P}$::$\text{EID}_\mathcal{P} := \text{EID}_\mathcal{P}$ // eids chosen by environment for preprocessing.
14: Π-TSSync2$_\mathcal{P}$::$\text{PP}_\mathcal{P} := st_{\text{pp}}$::$\text{PP}_\mathcal{P}$ // Preprocessings from st_{pp} (updated by Preprocess in $\mathcal{F}$-TSSync2)
15: **for** *proc* with $\mathcal{P} \in \text{Proc}[proc]$ and $\mathcal{P} \notin \Pi$-TSSync2::Proc[*proc*] **do**
16: $\quad\Pi$-TSSync2$_\mathcal{P}$::$\text{Proc}[proc] := \Pi$-TSSync2$_\mathcal{P}$::$\text{Proc}[proc] \cup \{\mathcal{P}\}$ // Enlist $\mathcal{P}$ for the process.
17: $\quad\Pi$-TSSync2$_\mathcal{P}$::$\text{Used}_\mathcal{P} := \Pi$-TSSync2$_\mathcal{P}$::$\text{Used}_\mathcal{P} \cup \text{EID}_\mathcal{P}[proc]$
18: $\quad(\boldsymbol{T}, m, ctx, (\mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}, ssid, round_{\text{start}}) := proc$ and $\mathsf{eid} := \text{EID}_\mathcal{P}[proc]$
19: $\quad\Pi$-TSSync2$_\mathcal{P}$::$\text{PM}_\mathcal{P}[\mathsf{eid}] := \mathbf{pm}_0 := (\mathsf{pp}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$

$\mathcal{S}_{\text{TSSync2}}$::RoundOK($\mathcal{P}$, *sid*) // Invoked by $\mathcal{G}$-Clock after $\mathcal{F}$-TSSync2::$\mathcal{P}$.Tick is called

20: **if** *sid* is the session of $\mathcal{F}$-TSSync2 **then** // $\mathcal{F}$-TSSync2::$\mathcal{P}$.Tick() has been called.
21: $\quad now := \mathcal{G}$-Clock::$\mathcal{S}_{\text{TSSync2}}$.GetRound()
22: $\quad(\text{Proc}, (\text{EID}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}, st_{\text{pp}}) := \mathcal{F}$-TSSync2::$\mathcal{S}_{\text{TSSync2}}$.GetProcessInfo()
23: $\quad$Set internal state of $\mathcal{F}$-TS3 to st_{pp}
24: $\quad$SyncState($\mathcal{P}$)
25: $\quad$// Execute the round for $\mathcal{P}$ (make it queue up messages etc.)
26: $\quad\Pi$-TSSync2$_\mathcal{P}$::$\mathcal{P}$.Tick(), but use $now := \text{RndDone}[\mathcal{P}] - 1$ in both Π-TSSync2$_\mathcal{P}$::$\mathcal{P}$.Tick and $\mathcal{F}$-NetSync::$\mathcal{P}$.Send // Ensures correct round is executed even if $\mathcal{G}$-Clock has advanced just before calling $\mathcal{S}_{\text{TSSync2}}$::RoundOK
27: $\quad$// Note that the line above causes $\mathcal{F}$-TS3::$\mathcal{P}$.Sign calls
28: $\quad$Let st'_{pp} be the updated internal state of $\mathcal{F}$-TS3 after the above calls
29: $\quad\mathcal{F}$-TSSync2::$\mathcal{S}_{\text{TSSync2}}$.SetPreprocState($st'_{\text{pp}}$)
30: **if** $\mathcal{G}$-Clock::$\mathcal{S}_{\text{TSSync2}}$.GetRound() has increased since the last $\mathcal{S}_{\text{TSSync2}}$::RoundOK($\cdot$) call **then**
31: $\quad$// Round has just ended! Prepare signatures for upcoming $\mathcal{F}$-TSSync2::GetSig calls.
32: $\quad$**for** $\mathcal{Q} \in \mathbf{P}$ **do**
33: $\quad\quad\Pi$-TSSync2$_\mathcal{Q}$::ReadRound($\mathcal{Q}$) // Retrieves this round's messages and updates internal state.
34: $\quad$**for** $(\mathcal{Q}, proc = (\cdot, round_{\text{start}}))$ with $\mathcal{Q} \in \text{Proc}[proc] \cap \mathbf{H}$ and $round_{\text{start}} + \mathsf{rnd}_{\text{sign}} = now$ **do**
35: $\quad\quad$// Signing process *proc* just finished.
36: $\quad\quad\mathsf{eid} := \text{EID}_\mathcal{Q}[proc]$
37: $\quad\quad\sigma \leftarrow \mathcal{F}$-TS3::Combine($\mathsf{pk}, \boldsymbol{T}, m, ctx, \text{PM}_\mathcal{Q}[\mathsf{eid}]$) // Compute signature according to Combine (the algorithm output by $\mathcal{A}$, not the procedure $\mathcal{F}$-TS3::Combine)
38: $\quad\quad\mathcal{F}$-TSSync2::$\mathcal{S}_{\text{TSSync2}}$.SetSig($\mathcal{Q}, proc, \sigma$) // Copy final signature to $\mathcal{F}$-TSSync2
39: $\mathcal{A}$::RoundOK($\mathcal{P}$, *sid*) // Invoke adversary, who is expecting this same call from $\mathcal{G}$-Clock

As in Theorem 3, the simulator runs its copy of $\mathcal{F}$-TS3 only for the purposes of preprocessing, signing, and to satisfy the adversary corrupting a party on $\mathcal{F}$-TS3. The main argument for the proof is about showing that verification in the real world (Π-TSSync2 / $\mathcal{F}$-TS3) behaves like the ideal world ($\mathcal{F}$-TS2).

One complication around the first bullet point above is that in the ideal world, $\mathcal{S}_{\text{TSSync2}}$ is not activated during preprocessing. This means that the simulator cannot directly use its $\mathcal{F}$-TS3 copy to simulate preprocessing. Instead, the simulator defines $\mathsf{Preprocess}_{\text{FTS3}}$ to exactly run $\mathcal{F}$-TS3's preprocessing, and it uses the state variable st_{pp} to keep track of the internal state of $\mathcal{F}$-TS3. This way, while the simulator is not aware of it, $\mathcal{F}$-TSSync2 executes preprocessing exactly as the internal copy of $\mathcal{F}$-TS3 would have executed it. When the simulator is activated again during TICK, it writes the updated state st_{pp} from $\mathcal{F}$-TSSync2 back to its internal $\mathcal{F}$-TS3 copy, keeping the state in sync for $\mathcal{F}$-TS3::$\mathcal{P}$.SIGN calls during TICK.

With that out of the way, the proof is mostly concerned with showing that VERIFY and GETSIG behave the same in the real and ideal world (as the other procedures are relatively straightforward invocations of Π-TSSync2 in both real and ideal world, as discussed above). The code of VERIFY is exactly the same in real and ideal world. For GETSIG, in the real world, $\mathcal{F}$-TS3::COMBINE is called, which computes Combine and then (if $\mathsf{ValidateC}_{\mathbf{t}}$ or $\mathsf{ValidateR}_{\mathbf{t}}$), ForceValidity. In the ideal world, the simulator computes Combine during ROUNDOK and then, if $\mathsf{ValidateCorR}_{\mathbf{t}}$, then $\mathcal{F}$-TSSync2 calls ForceValidity. The conditions of $\mathsf{ValidateC}_{\mathbf{t}} \vee \mathsf{ValidateR}_{\mathbf{t}}$ and $\mathsf{ValidateCorR}_{\mathbf{t}}$ are aligned (note that the channel is reliable, hence there is no case where honest parties disagree on the set of messages sent by honest parties), i.e., ForceValidity is called under the same conditions in both worlds.

It remains to show that the state variables read during GETSIG and VERIFY are the same between real and ideal world. Namely, these variables are pk, $\text{PM}_{\mathcal{P}}$, VER, and SGN.

- By design of the simulator, pk is set during initialization for both $\mathcal{F}$-TSSync2 in the ideal world and $\mathcal{F}$-TS3 in the real at the same time, once $\mathcal{A}$ outputs it.
- The protocol messages $\text{PM}_{\mathcal{P}}$ used to compute Combine are ultimately read from the network in ReadRound in the real world. In the ideal world, the simulator runs the network as-is, and reads the same messages during ROUNDOK. Hence $\text{PM}_{\mathcal{P}}$ is the same in both worlds.
- SGN is updated during SIGN for the new signing process in both real and ideal world in the same way.
- VER is written to during VERIFY in real and ideal world in the same way. Hence if VER is the same in both worlds before VERIFY, it will be the same after VERIFY as well. It is also written to during GETSIG via ForceValidity. However, as argued above, ForceValidity is called under the same conditions in both worlds with the same input, hence VER is updated in the same way in both worlds, too.

Overall, the state variables relevant for GETSIG and VERIFY are updated in the same way in both real and ideal world, and the same algorithms are called on them, hence the behavior of GETSIG and VERIFY is the same in both worlds. □

H $\mathcal{F}$-TSSync1 and Π-TSSync1

We list the protocol Π-TSSync1 realizing $\mathcal{F}$-TSSync1 in Protocol 6.

Theorem 6 (restated). The protocol Π-TSSync1$[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ (uniform-)UC-realizes $\mathcal{F}$-TSSync1$[\mathbf{P}, \mathsf{rnd}_{\text{sign}}, \mathbf{t}, \mathsf{cor}]$ in the $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl} = \text{UF-1}, \mathbf{rnd} = (\mathsf{rnd}_{\text{T}} = 1, \mathsf{rnd}_{\text{msg}} = 1, \mathsf{rnd}_{\text{ctx}} = 1, \mathsf{rnd}_{\text{com}} = 1, \mathsf{rnd}_{\text{sign}}), \mathsf{ssid} = (\cdot \mapsto 0), \mathsf{cor}]$ and $\mathcal{F}$-NetSync hybrid model in the presence of $\mathcal{G}$-Clock.

The proof is analogous to the proof of Theorem 5 (listed in Appendix G), minus the preprocessing.

I Example Instantiations

I.1 FROST1

FROST1 is a two-round protocol for threshold Schnorr signatures. We consider the same threshold for correctness, corruption, and hierarchy, i.e., $t_{\text{correct}} = t_{\text{corrupt}} = t_{\text{UF}}$ and no robustness, i.e., $t_{\text{robust}} = |\mathbf{P}| + 1$. As $\mathsf{rnd}_{\text{com}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{ctx}} = 2$, the SSID is simply the collection of all $\mathsf{pm}_{\mathcal{Q},1}$, i.e.,

Protocol 8: Π-TSSync1$_{\mathcal{P}}[\mathbf{P}, \mathbf{t}, \mathsf{cor}]$ realizing $\mathcal{F}$-TSSync1

With access to $\mathcal{F}$-TS3$[\mathbf{P}, \mathbf{t}, \mathsf{lvl} = \text{UF-1}, \mathbf{rnd} = (\mathsf{rnd}_{\text{T}} = 1, \mathsf{rnd}_{\text{msg}} = 1, \mathsf{rnd}_{\text{ctx}} = 1, \mathsf{rnd}_{\text{com}} = 1, \mathsf{rnd}_{\text{sign}}), \mathsf{ssid} = (\cdot \mapsto 0), \mathsf{cor}]$ (Functionality 4) and $\mathcal{F}$-NetSync (Functionality 8) and $\mathcal{G}$-Clock (Functionality 7).

Init()

Proc$[\cdot] := \varnothing$
PM$_{\mathcal{P}}[\cdot] :=$ () for all $\mathcal{P} \in \mathbf{P}$
RndDone$[\cdot] := 0$
$\mathsf{pk} := \bot$
($\clubsuit$)$\mathcal{F}$-TS3.Init()

$\mathcal{P}$.KeyGen() if $\mathcal{P} \in \mathbf{P}$

return $\mathsf{pk} := \mathcal{F}$-TS3::$\mathcal{P}$.KeyGen()

$\mathcal{A}$.Corrupt()

assert $\mathcal{P} \in \mathbf{P}$
assert $\mathsf{cor} =$ adaptive $\vee$ [$\mathcal{P}$.KeyGen has not been called yet]
$\mathbf{C} := \mathbf{C} \cup \{\mathcal{P}\}$, $\mathbf{H} := \mathbf{H} \setminus \{\mathcal{P}\}$
return (Proc, PM$_{\mathcal{P}}$, RndDone, pk)

$\mathcal{P}$.Verify(pk', m, σ)

return $\mathcal{F}$-TS3::$\mathcal{P}$.Verify(pk', m, σ)

$\mathcal{P}$.Sign($\boldsymbol{T}, m, ctx$) if $\mathcal{P} \in \mathbf{P}$

assert $\mathcal{P}$.KeyGen() has been called.
$\mathcal{G}$-Clock::$\mathcal{P}$.Register() // Ensure honest parties wait on $\mathcal{P}$ to finish round.
assert $\boldsymbol{T} \subseteq \mathbf{P} \wedge \mathcal{P} \in \boldsymbol{T}$
$now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
assert $now >$ RndDone$[\mathcal{P}]$ // $\mathcal{P}$.Tick() not yet called this round.
$round_{\text{start}} := now$ // Start this round.
$proc := (\boldsymbol{T}, m, ctx, round_{\text{start}})$
Proc$[proc] :=$ Proc$[proc] \cup \{\mathcal{P}\}$
$\mathsf{inp} := (\boldsymbol{T}, m, ctx)$
$\mathcal{F}$-TS3::$\mathcal{P}$.Sign$_2$(eid := $(proc, \text{dummy})$, inp) // Sole purpose: make $\mathcal{F}$-TS3 add $\mathcal{P}$ to SGN$[m, \cdot]$ already.

$\mathcal{P}$.GetSig($proc$) if $\mathcal{P} \in \mathbf{P}$

ReadRound($\mathcal{P}$) // Read latest network msgs
$now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
$(\boldsymbol{T}, m, ctx, round_{\text{start}}) := proc$
assert $\mathcal{P} \in$ Proc$[proc]$ // $\mathcal{P}$ participated.
assert $now \geq round_{\text{start}} + \mathsf{rnd}_{\text{sign}}$ // Finished
$\sigma \leftarrow$ $\mathcal{F}$-TS3::$\mathcal{P}$.Combine(pk, $\boldsymbol{T}, m, ctx,$ PM$_{\mathcal{P}}[proc]$)
return σ

($\clubsuit$)$\mathcal{P}$.Tick() if $\mathcal{P} \in \mathbf{P}$ // Send messages and finish round

$now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
assert RndDone$[\mathcal{P}] < now$
RndDone$[\mathcal{P}] := now$
ReadRound($\mathcal{P}$) // Read latest messages from network, update finished signatures
for $proc := (\boldsymbol{T}, m, ctx, round_{\text{start}})$ with $\mathcal{P} \in$ Proc$[proc]$ **do**
  $round := now - round_{\text{start}} + 1$ // Sign$_{round}$ to be called in this round.
  if $round \leq \mathsf{rnd}_{\text{sign}}$ **then** // Still rounds to execute.
    $\mathsf{inp} := (\boldsymbol{T}, m, ctx,$ PM$_{\mathcal{P}}[proc])$
    $\mathsf{pm}' \leftarrow$ $\mathcal{F}$-TS3::$\mathcal{P}$.Sign$_{round}$(eid := $proc$, inp)
    for $\mathcal{R} \in \boldsymbol{T}$ **do**
      $\mathcal{F}$-NetSync::$\mathcal{P}$.Send($\mathcal{R}, port := proc, \mathsf{pm}'$)
($\clubsuit$)$\mathcal{G}$-Clock::$\mathcal{P}$.RoundOK()

ReadRound($\mathcal{P}$) (internal) // Receives messages and updates internal data structures. Called at least once every round for each honest user (by Tick). Safe to call multiple times per round.

$now :=$ $\mathcal{G}$-Clock::$\mathcal{P}$.GetRound()
for $proc := (\boldsymbol{T}, m, ctx, round_{\text{start}})$ with $\mathcal{P} \in$ Proc$[proc]$ **do**
  $round := now - round_{\text{start}} + 1$ // Current protocol round (receiving messages from $round - 1$)
  if $round = 1 \vee round > \mathsf{rnd}_{\text{sign}} + 1$ **then** skip below and continue for loop with next $proc$
  $(\mathbf{pm}_k)_{k=1}^{\ell} :=$ PM$_{\mathcal{P}}[proc]$ // Read previously received messages. $\ell \in \{round - 2, round - 1\}$
  $\mathbf{pm}_{round-1} :=$ $\mathcal{F}$-NetSync::$\mathcal{P}$.Receive($port := proc$)
  $\mathbf{pm}_{round-1} := (\mathbf{pm}_{round-1}[\mathcal{Q}])_{\mathcal{Q} \in \boldsymbol{T}}$ // Filter $\mathbf{pm}_{round-1}$ for $\boldsymbol{T}$.
  PM$_{\mathcal{P}}[proc] := (\mathbf{pm}_k)_{k=1}^{round-1}$ // Update PM$_{\mathcal{P}}[proc]$ with $\mathbf{pm}_{round-1}$

$(\boldsymbol{T}, ctx, (R_\mathcal{Q}, S_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}})$. It is easy to check that Protocol 9 satisfies our correctness definition perfectly. That is, as long as all parties in $\boldsymbol{T}$ receive the same $\boldsymbol{T}$, m, $(\mathsf{pm}_{\mathcal{Q},1})_{\mathcal{Q}\in\boldsymbol{T}} = (R_\mathcal{Q}, S_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$, then they all output the same R after Sign_2. Since Sign_2 also outputs $z_\mathcal{Q}$ derived from the same $c = \mathsf{H}(\mathsf{pk}, m, R)$, the combined z passes verification with respect to R, pk, and m.

Protocol 9: FROST1 [KG20]

$\underline{\mathsf{Setup}(1^\kappa)}$
1: **return** $\mathsf{par} = (\mathbb{G}, q, g, \mathsf{H})$

$\underline{\mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})}$ with $\mathbf{t} = (t_{\text{correct}}, t_{\text{corrupt}}, t_{\text{UF}}, t_{\text{robust}})$
2: $(\mathbb{G}, q, g, \mathsf{H}) := \mathsf{par}$
3: **for** $i = 0, \ldots, t_{\text{corrupt}} - 1$ **do**
4: $\quad a_i \xleftarrow{\$} \mathbb{Z}_q$
5: $f(x) := a_0 + a_1 x + \ldots + a_{t_{\text{corrupt}}-1} x^{t_{\text{corrupt}}-1}$
6: $i = 1$
7: **for** $\mathcal{P} \in \mathbf{P}$ **do**
8: $\quad \mathsf{sk}_\mathcal{P} := f(i)$
9: $\quad \mathsf{pk}_\mathcal{P} := g^{\mathsf{sk}_\mathcal{P}}$
10: $\quad i = i + 1$
11: $\mathsf{pk} = g^{a_0}$
12: **return** $(\mathsf{pk}, \{\mathsf{sk}_\mathcal{P}\}_{\mathcal{P}\in\mathbf{P}})$

$\underline{\mathsf{Verify}(\mathsf{pk}, m, \sigma)}$
1: $(R, z) := \sigma$
2: $c \leftarrow \mathsf{H}(\mathsf{pk}, m, R)$
3: **assert** $g^z = R \cdot \mathsf{pk}^c$
4: **return** 1

$\underline{\mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\text{com}}-1]})}$ // $\mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{ctx}} = \mathsf{rnd}_{\text{com}} = 2$
5: $(R_Q, S_Q)_{Q\in\boldsymbol{T}} := \mathbf{pm}_1$
6: **return** $(\boldsymbol{T}, ctx, (R_Q, S_Q)_{Q\in\boldsymbol{T}})$

$\underline{\mathsf{Sign}_1(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \bot, \bot, \bot, \bot)}$
7: $(\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P}) := \mathsf{st}_\mathcal{P}$
8: $r \xleftarrow{\$} \mathbb{Z}_q$; $s \xleftarrow{\$} \mathbb{Z}_q$
9: $\mathsf{pm}' := (R, S) = (g^r, g^s)$
10: $\mathsf{st}'_{\mathcal{P},\mathsf{eid}} := (r, s)$
11: **return** $(\mathsf{st}_\mathcal{P}, \mathsf{st}'_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}')$

$\underline{\mathsf{Sign}_2(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \mathbf{pm}_1)}$
12: $(\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P}) := \mathsf{st}_\mathcal{P}$
13: $(r, s) := \mathsf{st}_{\mathcal{P},\mathsf{eid}}$
14: $(R_\mathcal{Q}, S_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}} := \mathbf{pm}_1$
15: **for** $\mathcal{Q} \in \boldsymbol{T}$ **do**
16: $\quad d_\mathcal{Q} \leftarrow \mathsf{H}(\mathsf{pk}, \boldsymbol{T}, m, \mathbf{pm}_1, \mathcal{Q})$
17: $R := \prod_{\mathcal{Q}\in\boldsymbol{T}} R_\mathcal{Q} \cdot S_\mathcal{Q}^{d_\mathcal{Q}}$
18: $c \leftarrow \mathsf{H}(\mathsf{pk}, m, R)$
19: $z_\mathcal{P} := r_\mathcal{P} + d_\mathcal{P} \cdot s_\mathcal{P} + c \cdot \lambda_{\boldsymbol{T},\mathcal{P}} \cdot \mathsf{sk}_\mathcal{P}$
20: $\mathsf{pm}' := z_\mathcal{P}$
21: **return** $(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}')$

$\underline{\mathsf{Combine}(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_1, \mathbf{pm}_2))}$
22: $(R_\mathcal{Q}, S_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}} := \mathbf{pm}_1$
23: $(z_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}} := \mathbf{pm}_2$
24: $z := 0$
25: **for** $\mathcal{Q} \in \boldsymbol{T}$ **do**
26: $\quad d_\mathcal{Q} \leftarrow \mathsf{H}(\mathsf{pk}, \boldsymbol{T}, m, \mathbf{pm}_1, \mathcal{Q})$
27: $\quad z := z + z_\mathcal{Q}$
28: $R := \prod_{\mathcal{Q}\in\boldsymbol{T}} R_\mathcal{Q} \cdot S_\mathcal{Q}^{d_\mathcal{Q}}$
29: **return** (R, z)

I.2 Threshold Raccoon

Threshold Raccoon is a three-round protocol generating Raccoon signatures [dPKPR24] in a threshold manner. In Protocol 10, we present a simplified version of Threshold Raccoon presented in [KRT24], operating on a ring of polynomials $\mathcal{R} = \mathbb{Z}[X]/(X^n + 1)$ and the corresponding quotient ring $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. All the notations except party indices and thresholds follow the original paper [KRT24] and we refer readers to the paper for details. Similar to FROST, we consider $t_{\text{correct}} = t_{\text{corrupt}} = t_{\text{UF}}$ and $t_{\text{robust}} = |\mathbf{P}| + 1$. We consider $\mathsf{rnd}_{\text{com}} = \mathsf{rnd}_{\text{T}} = \mathsf{rnd}_{\text{msg}} = \mathsf{rnd}_{\text{ctx}} = 1$ assuming no preprocessing and the SSID is simply the collection of all $\mathsf{pm}_{\mathcal{Q},1}$, i.e., $(\boldsymbol{T}, ctx, (\mathsf{cmt}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}})$. Unlike FROST, Threshold Raccoon is only statistically correct, i.e., it may fail with negligible probability even if all parties are honest.

Lemma 3. *For* $\mathsf{cor} \in \{\text{adaptive}, \text{static}\}$*, Threshold Raccoon described in Protocol 10 is* $(\mathsf{cor}, \mathbf{P}, \mathbf{t})$*-correct if* $\nu_w \geq 4$*, the adversary only makes polynomially many queries to* $(\mathcal{O}_{\text{SIGN},k})_{k\in[\mathsf{rnd}_{\text{sign}}]}$ *and the random oracle* $\mathsf{H}_{\mathsf{com}}$*, and:*

$$B = e^{1/4} \cdot (W\sigma_t + \sqrt{t_{\text{correct}}}\,\sigma_w)\sqrt{n(k+\ell)} + (W \cdot 2^{\nu_t} + 2^{\nu_w+1}) \cdot \sqrt{nk}.$$

Proof. From Lemma 3.1 of [KRT24], assuming B and ν_w satisfy the conditions stated, the probability that any adversary wins the $\mathsf{1TCOR}$ game is negligible. We argue TCOR essentially reduces to $\mathsf{1TCOR}$ as long as the adversary makes polynomially many queries to $(\mathcal{O}_{\text{SIGN},k})_{k\in[\mathsf{rnd}_{\text{sign}}]}$ and $\mathsf{H}_{\mathsf{com}}$. Let Hyb_1 be the same as $\mathsf{Exp}^{\mathsf{cor},\mathsf{TCOR}}_{\Sigma_{\text{TS}},\mathcal{A}}(1^\kappa,\mathbf{P},\mathbf{t})$. We consider the following hybrid experiment Hyb_2, which is identical to Hyb_1 except the following changes:

- If $\mathcal{O}_{\text{SIGN},1}$ is called for party $\mathcal{P}$ and eid, instead of running Sign_1 honestly, the oracle samples $\mathsf{cmt}_{\mathcal{P}} \in \{0,1\}^{2\kappa}$ uniformly at random.
- If $\mathcal{O}_{\text{SIGN},2}$ is called for the same pair of party $\mathcal{P}$ and eid, instead of running Sign_2 honestly, the oracle computes $\mathbf{w}_{\mathcal{P}} \in \mathcal{R}_q^k$ as in Sign_1, programs $\mathsf{H}_{\mathsf{com}}(\mathcal{P},\mathbf{w}_{\mathcal{P}}) = \mathsf{cmt}_{\mathcal{P}}$, and returns $\mathbf{w}_{\mathcal{P}}$. If $\mathsf{H}_{\mathsf{com}}(\mathcal{P},\mathbf{w}_{\mathcal{P}})$ has already been defined, the oracle aborts and returns $\perp$.

Note that this hybrid is analogous to Game_2 of Theorem 4.1 of [KRT24], where they also program the random oracle $\mathsf{H}_{\mathsf{com}}$ to ensure that the commitments $\mathsf{cmt}_{\mathcal{P}}$ are uniformly random and independent for each pair of $\mathcal{P}$ and eid. Theorem 4.1 of [KRT24] implies that the probability that $\mathcal{O}_{\text{SIGN},2}$ aborts in Hyb_2 is negligible in κ by high min-entropy of $\mathbf{w}_{\mathcal{P}}$, if the adversary makes only polynomially many queries to $(\mathcal{O}_{\text{SIGN},k})_{k\in[\mathsf{rnd}_{\text{sign}}]}$ and $\mathsf{H}_{\mathsf{com}}$.

The probability that the adversary wins Hyb_2 is bounded via the $\mathsf{1TCOR}$ property of Threshold Raccoon: since the commitments $\mathsf{cmt}_{\mathcal{P}}$ are uniformly random and independent, given fixed $\boldsymbol{T}$, for any choice of execution identifier $\mathsf{eid}_{\mathcal{P}}$ for each party $\mathcal{P} \in \boldsymbol{T}$, the probability that $\mathsf{Combine}$ outputs an invalid signature on faithfully generated transcripts as input is negligible in κ. Since the adversary makes only polynomially many signing queries, the probability that any signing session among uncorrupted parties yields an invalid signature is negligible in κ by a union bound. The same argument holds for the adaptive corruption case, since the winning condition of TCOR requires the adversary to specify a signer set $\boldsymbol{T}$ consisting entirely of uncorrupted parties. □

Variants of Threshold Raccoon. We can apply the above argument to show that the variants of Threshold Raccoon presented in [KRT24] also satisfy TCOR:

- T-Raccoon-KRT-5-rnd [KRT24] is a five-round variant of Threshold Raccoon, where the first round protocol messages are uniformly random strings generated by each party, which are used to derive one-time randomness to mask $\mathbf{w}_{\mathcal{P}}$. The second round is analogous to Sign_1 of Threshold Raccoon, where each party generates $\mathbf{w}_{\mathcal{P}}$ and commits to masked $\mathbf{w}_{\mathcal{P}}$ via a random oracle $\mathsf{H}_{\mathsf{com}}$. In the third round, each party generates a (non-threshold) signature on the transcript of the first two rounds. The fourth and fifth rounds are analogous to Sign_2 and Sign_3 of Threshold Raccoon. $\mathsf{1TCOR}$ of T-Raccoon-KRT-5-rnd is shown in Lemma 5.1 of [KRT24] under the same conditions on B and ν_w as Threshold Raccoon. As T-Raccoon-KRT-5-rnd can also defer the generation of $\mathbf{w}_{\mathcal{P}}$ to the fourth round, the same hybrid argument as above applies to show that T-Raccoon-KRT-5-rnd also satisfies TCOR.
- T-Raccoon-KRT-4-rnd [KRT24] is a four-round variant of Threshold Raccoon, which is essentially the same as T-Raccoon-KRT-5-rnd except that the uniformly random strings generated in the first round are replaced with a pre-agreed session ID. The TCOR property of T-Raccoon-KRT-4-rnd is shown via the same argument as T-Raccoon-KRT-5-rnd.
- T-Raccoon-IA [PKN+25] is a variant of Threshold Raccoon with additional dispute resolution rounds, allowing parties to identify and exclude misbehaving parties from the signer set $\boldsymbol{T}$ in case of a failed signing session. If all parties in $\boldsymbol{T}$ are honest, the dispute resolution rounds are not triggered and the signing session proceeds as in Threshold Raccoon, so TCOR of T-Raccoon-IA also follows from the same argument as above.
- Finally [BCdP+25] and Mithril [CdPE+26][11] are three-round protocols with the same commit-open-sign structure as Threshold Raccoon, except that each party locally performs rejection sampling to ensure that $\mathbf{z}_{\mathcal{P}}$ is independent of the secret key share $\mathbf{s}_{\mathcal{P}}$. Since each run of the signing protocol fails with non-negligible probability due to rejection sampling, one can run sufficiently many parallel signing sessions to reduce the error probability in the $\mathsf{1TCOR}$ game to negligible. Under this execution model, the same hybrid argument as above shows that these two protocols also satisfy TCOR.

[11] Note that the protocol also comes with DKG. Since we do not consider UC modelling of DKG in this work, we assume Mithril with trusted key generation.

Protocol 10: Threshold Raccoon [KRT24]

$\underline{\mathsf{Setup}(1^\kappa)}$

$\mathbf{A} \xleftarrow{\$} \mathcal{R}_q^{k\times\ell}$
$\mathsf{par} := (\mathbf{A}, N, t_{\mathrm{correct}})$
return par

$\underline{\mathsf{KeyGen}(\mathsf{par}, \mathbf{P}, \mathbf{t})}$ with $\mathbf{t} = (t_{\mathrm{correct}}, t_{\mathrm{corrupt}}, t_{\mathrm{UF}}, t_{\mathrm{robust}})$

$(\mathbf{s}, \mathbf{e}) \xleftarrow{\$} \mathcal{D}^\ell_{\sigma_{\mathbf{t}}} \times \mathcal{D}^k_{\sigma_{\mathbf{t}}}$
$\mathbf{t} := [\mathbf{As} + \mathbf{e}]_{\nu_t} \in \mathcal{R}^k_{q_{\nu_t}}$
for $\mathcal{P} \in \mathbf{P}$ **do**
  for $\mathcal{Q} \in \mathbf{P}$ **do**
    $\mathsf{rand}_{\mathcal{P},\mathcal{Q}} \xleftarrow{\$} \{0,1\}^\kappa$
    $\mathsf{seed}_{\mathcal{P},\mathcal{Q}} := \mathcal{P} \| \mathcal{Q} \| \mathsf{rand}_{\mathcal{P},\mathcal{Q}}$
$(\mathbf{seed}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}} := ((\mathsf{seed}_{\mathcal{P},\mathcal{Q}}, \mathsf{seed}_{\mathcal{Q},\mathcal{P}})_{\mathcal{Q}\in\mathbf{P}})_{\mathcal{P}\in\mathbf{P}}$
$\tilde{P} \xleftarrow{\$} \mathcal{R}^\ell_q[X]$ s.t. $\deg(\tilde{P}) = t_{\mathrm{correct}} - 1$, $\tilde{P}(0) = \mathbf{s}$
$(\mathbf{s}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}} := (\tilde{P}(\mathcal{P}))_{\mathcal{P}\in\mathbf{P}}$
$\mathsf{pk} := (\mathsf{par}, \mathbf{t})$
$(\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}} := (\mathbf{s}_\mathcal{P}, \mathbf{seed}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}}$
return $(\mathsf{pk}, (\mathsf{sk}_\mathcal{P})_{\mathcal{P}\in\mathbf{P}})$

$\underline{\mathsf{Verify}(\mathsf{pk}, m, \sigma)}$

$(c, \mathbf{z}, \mathbf{h}) := \sigma$
$c' := \mathsf{H}_c(\mathsf{pk}, m, [\mathbf{Az} - 2^{\nu_t} \cdot c \cdot \mathbf{t}]_{\nu_w} + \mathbf{h})$
assert $c = c' \wedge \|(\mathbf{z}, 2^{\nu_w} \cdot \mathbf{h})\|_2 \le B$
return 1

$\underline{\mathsf{ssid}(\boldsymbol{T}, m, ctx, (\mathbf{pm}_k)_{k\in[\mathsf{rnd}_{\mathrm{com}}-1]})}$ // $\mathsf{rnd}_{\mathrm{T}} = \mathsf{rnd}_{\mathrm{msg}} = \mathsf{rnd}_{\mathrm{ctx}} = \mathsf{rnd}_{\mathrm{com}} = 1$

$(\mathsf{cmt}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}} := \mathbf{pm}_1$
return $(\boldsymbol{T}, ctx, (\mathsf{cmt}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}})$

$\underline{\mathsf{Sign}_1(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \bot)}$

$(\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P}) := \mathsf{st}_\mathcal{P}$
$(\mathbf{r}_\mathcal{P}, \mathbf{e}'_\mathcal{P}) \xleftarrow{\$} \mathcal{D}^\ell_{\sigma_{\mathbf{w}}} \times \mathcal{D}^k_{\sigma_{\mathbf{w}}}$
$\mathbf{w}_\mathcal{P} := \mathbf{Ar}_\mathcal{P} + \mathbf{e}'_\mathcal{P} \in \mathcal{R}^k_q$
$\mathsf{cmt}_\mathcal{P} := \mathsf{H}_{\mathsf{com}}(\mathcal{P}, \mathbf{w}_\mathcal{P}) \in \{0,1\}^{2\kappa}$
$\mathsf{st}'_{\mathcal{P},\mathsf{eid}} := (\mathsf{cmt}_\mathcal{P}, \mathbf{w}_\mathcal{P}, \mathbf{r}_\mathcal{P})$
return $(\mathsf{st}_\mathcal{P}, \mathsf{st}'_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}_{\mathcal{P},1} := \mathsf{cmt}_\mathcal{P})$

$\underline{\mathsf{Sign}_2(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \mathbf{pm}_1)}$

$(\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P}) := \mathsf{st}_\mathcal{P}$
$(\mathsf{cmt}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}} := \mathbf{pm}_1$
$(\mathsf{cmt}_\mathcal{P}, \mathbf{w}_\mathcal{P}, \mathbf{r}_\mathcal{P}) := \mathsf{st}_{\mathcal{P},\mathsf{eid}}$
$\mathsf{st}_{\mathcal{P},\mathsf{eid}} := (\boldsymbol{T}, m, (\mathsf{cmt}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}, \mathbf{w}_\mathcal{P}, \mathbf{r}_\mathcal{P})$
return $(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}_{\mathcal{P},2} := \mathbf{w}_\mathcal{P})$

$\underline{\mathsf{Sign}_3(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \boldsymbol{T}, m, ctx, \mathbf{pm}_2)}$

$(\mathcal{P}, \mathsf{pk}, \mathsf{sk}_\mathcal{P}) := \mathsf{st}_\mathcal{P}$
$(\mathbf{s}_\mathcal{P}, \mathbf{seed}_\mathcal{P}) := \mathsf{sk}_\mathcal{P}$
$(\mathbf{w}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}\setminus\{\mathcal{P}\}} := (\mathsf{pm}_{\mathcal{Q},2})_{\mathcal{Q}\in\boldsymbol{T}\setminus\{\mathcal{P}\}}$
$(\boldsymbol{T}, m, (\mathsf{cmt}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}, \mathbf{w}_\mathcal{P}, \mathbf{r}_\mathcal{P}) := \mathsf{st}_{\mathcal{P},\mathsf{eid}}$
assert $\forall \mathcal{Q} \in \boldsymbol{T},\ \mathsf{cmt}_\mathcal{Q} = \mathsf{H}_{\mathsf{com}}(\mathcal{Q}, \mathbf{w}_\mathcal{Q})$
$\mathsf{ctnt}_z := \boldsymbol{T} \| m \| (\mathsf{cmt}_\mathcal{Q}, \mathbf{w}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}}$
$\mathbf{w} := \left[\sum_{\mathcal{Q}\in\boldsymbol{T}} \mathbf{w}_\mathcal{Q}\right]_{\nu_w} \in \mathcal{R}^k_{q_{\nu_w}}$
$c := \mathsf{H}_c(\mathsf{pk}, m, \mathbf{w})$
$\Delta_\mathcal{P} := \mathsf{ZeroShare}(\mathbf{seed}_\mathcal{P}[\boldsymbol{T}], \mathsf{ctnt}_z) \in \mathcal{R}^\ell_q$
$\mathbf{z}_\mathcal{P} := c \cdot L_{\boldsymbol{T},\mathcal{P}} \cdot \mathbf{s}_\mathcal{P} + \mathbf{r}_\mathcal{P} + \Delta_\mathcal{P} \in \mathcal{R}^\ell_q$
return $(\mathsf{st}_\mathcal{P}, \mathsf{st}_{\mathcal{P},\mathsf{eid}}, \mathsf{pm}_{\mathcal{P},3} := \mathbf{z}_\mathcal{P})$

$\underline{\mathsf{Combine}(\mathsf{pk}, \boldsymbol{T}, m, ctx, (\mathbf{pm}_1, \mathbf{pm}_2, \mathbf{pm}_3))}$

$(\mathsf{par}, \mathbf{t}) := \mathsf{pk}$
$(\mathbf{w}_\mathcal{Q}, \mathbf{z}_\mathcal{Q})_{\mathcal{Q}\in\boldsymbol{T}} := (\mathbf{pm}_2, \mathbf{pm}_3)$
$\mathbf{w} := \left[\sum_{\mathcal{Q}\in\boldsymbol{T}} \mathbf{w}_\mathcal{Q}\right]_{\nu_w}$
$\mathbf{z} := \sum_{\mathcal{Q}\in\boldsymbol{T}} \mathbf{z}_\mathcal{Q} \in \mathcal{R}^\ell_q$
$c := \mathsf{H}_c(\mathsf{pk}, m, \mathbf{w})$
$\mathbf{y} := [\mathbf{Az} - 2^{\nu_t} \cdot c \cdot \mathbf{t}]_{\nu_w} \in \mathcal{R}^k_{q_{\nu_w}}$
$\mathbf{h} := \mathbf{w} - \mathbf{y} \in \mathcal{R}^k_{q_{\nu_w}}$
return $\sigma := (c, \mathbf{z}, \mathbf{h})$

J Glossary of boxed items

Functionalities

Functionality 1: $\mathcal{F}$-Net . . . 9
Functionality 2: $\mathcal{G}$-Setup[Setup] . . . 9
Functionality 3: $\mathcal{F}$-Gen[$\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}$::Setup, Σ_{TS}::KeyGen] . . . 10
Functionality 4: Fine-grained $\mathcal{F}$-TS3[$\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathbf{rnd}, \mathsf{ssid}, \mathsf{cor}$] . . . 19
Functionality 5: Moderate $\mathcal{F}$-TS2[$\mathbf{P}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}$] . . . 24
Functionality 6: Simple $\mathcal{F}$-TS1[$\mathbf{P}, \mathbf{t}, \mathsf{cor}$] . . . 28
Functionality 7: $\mathcal{G}$-Clock . . . 29
Functionality 8: $\mathcal{F}$-NetSync[$\mathbf{P}$] . . . 29
Functionality 9: $\mathcal{F}$-TSSync2[$\mathbf{P}, \mathsf{rnd}_{\mathrm{sign}}, \mathbf{t}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}$] . . . 31
Functionality 10: $\mathcal{F}$-TSSync1[$\mathbf{P}, \mathsf{rnd}_{\mathrm{sign}}, \mathbf{t}, \mathsf{cor}$] . . . 33
Functionality 11: $\mathcal{G}$-RO[ℓ] . . . 34
Functionality 12: $\mathcal{F}$-Sig[$\mathcal{P}^*$] . . . 40

Protocols

Protocol 1: Π-TS3$_{\mathcal{P}}$[$\mathbf{P}, \mathbf{t}, \Sigma_{\mathsf{TS}}, \mathsf{cor}$] realizing $\mathcal{F}$-TS3 . . . 23
Protocol 2: Hybrid Π-TS3-1 . . . 42
Protocol 3: Hybrid Π-TS3-2 . . . 42
Protocol 4: Hybrid Π-TS3-3 . . . 44
Protocol 5: Π-TS2$_{\mathcal{P}}$[$\mathbf{P}, \mathbf{rnd}, \mathsf{cor}$] realizing $\mathcal{F}$-TS2 . . . 47
Protocol 6: Π-TS1$_{\mathcal{P}}$[$\mathbf{P}, \mathbf{t}, \mathsf{cor}$] realizing $\mathcal{F}$-TS1 . . . 48
Protocol 7: Π-TSSync2$_{\mathcal{P}}$[$\mathbf{P}, \mathsf{rnd}_{\mathrm{sign}}, \mathsf{lvl}, \mathsf{ssid}, \mathsf{cor}$] realizing $\mathcal{F}$-TSSync2 . . . 50
Protocol 8: Π-TSSync1$_{\mathcal{P}}$[$\mathbf{P}, \mathbf{t}, \mathsf{cor}$] realizing $\mathcal{F}$-TSSync1 . . . 53
Protocol 9: FROST1 [KG20] . . . 54
Protocol 10: Threshold Raccoon [KRT24] . . . 56

Simulators

Simulator 1: $\mathcal{S}_{\mathrm{TS3}}$ . . . 41
Simulator 2: $\mathcal{S}_{\mathrm{TS2}}$ . . . 45
Simulator 3: $\mathcal{S}_{\mathrm{TS1}}$ . . . 49
Simulator 4: $\mathcal{S}_{\mathrm{TSSync2}}$ . . . 51

Games

Game 1: Correctness and Robustness Experiments . . . 13
Game 2: One-time Correctness Experiment . . . 14
Game 3: Unforgeability experiment for a threshold signature scheme . . . 15

Oracles

Oracle 1: Oracles for Signing and Corruption . . . 15